DEVELOPMENT OF VIRTUAL REALITY SOFTWARE FOR THE DESIGN OF SPATIAL COMPLIANT MECHANISMS ON OCULUS RIFT DEVICE

BY

SUDHAMS (RAMKUMAR) KOMANDURI RANGANATH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Systems and Entrepreneurial Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Assistant Professor Girish Krishnan

ABSTRACT

The objective of this study was to develop a Virtual Reality (VR) design software platform that enables proper three-dimensional visualization of some of the design guidelines for compliant mechanisms, that were challenging to implement load flow visualization method on pen and paper. Some of the challenges include the three-dimensional visualization of the truncated hemispherical band (intersection between the modified input and output hemispherical bands), freedom ray and constraint plane. A three-dimensional VR design platform has been established for designing Spatial Compliant Mechanism (SCM) using load flow visualization method in this thesis research.

To achieve this goal, the approach was split into five stages: development of architecture for creating essential features of the design software; development of VR software algorithm through the application of Unity3D assets; testing and validation of the software by checking if the three-dimensional design guidelines, using Load Flow Visualization method, can be implemented for modeling SCM; enabling the user to move around this VR environment and getting an immersive experience of design modelling and analysis; performing trials of simple SCM design modeling on the VR software.

The architectural framework necessary for the creation and execution of various key features of the design software was developed and validated. The VR software was finally validated through the application of five single input-single output compliant mechanism designs. All the five designs were found to be compatible with the three-dimensional design guidelines using load flow visualization method. The key inferences of this validation include:

a) For the same design problem of a SCM, there can be multiple alternate solutions by changing the number of transmitters and/or intermediate points.

b) The deflection direction of intermediate point depends on the orientation of the constraint plane with respect to the point.

To my parents, Siddharth, Ammulu and Allison

ACKNOWLEDGEMENTS

I strongly feel that meeting my research advisor Professor Krishnan was part of my destiny because every discussion with him used to be a great learning experience. He gave a lot of creative freedom and kindly let me come up with interesting research ideas. I also learnt how to take pragmatic and smart decisions, while facing challenges posed by life. He has also been very kind and supportive when I was going through some health issues.

I would like to whole-heartedly appreciate Sree Kalyan Patiballa, my research colleague, for bringing my thesis to the present form. He constantly kept reminding me about the motivation behind our research project and thus helped me focus on solving it single-mindedly. He also regularly reviewed my thesis and gave invaluable feedback for improving my technical writing skills. Another research colleague, Kazuhiro Uchikata, was a great team player and we used to have exciting brainstorming sessions. Our passion for Virtual Reality enabled us to complete the software development in a short period of time. Kazu's contribution to the research was commendable.

This journey would not have been possible without the moral support from my parents, brother and Dr. Jones. In fact, the credit to all my successes will go to my family because they were the ones who made me bounce back from failures in the past. Dr. Jones was very caring and made my stay at Champaign as comfortable as my hometown. My biggest takeaway from UIUC is that I have become a stronger person and I am ready to go out and solve some real-world problems.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION 1

CHAPTER 2: VR DESIGN SOFTWARE.....7

CHAPTER 3: REVIEW OF LOAD FLOW VISUALIZATION 31

CHAPTER 4: DESIGNING SCM USING VR SOFTWARE.....40

CHAPTER 5: CONCLUSIONS AND RECOMMENDATIONS.....61

APPENDIX A: SOURCE CODE DESCRIPTION OF VR SOFTWARE 65

APPENDIX B: COPYRIGHT PERMISSION [UNITY] 98

APPENDIX C: COPYRIGHT PERMISSION [ASME] 99

REFERENCES 100

CHAPTER 1: INTRODUCTION

1.1 DESIGN METHODS FOR COMPLIANT MECHANISMS

While conventional mechanical devices comprise of rigid links and joints, an individual flexible mechanism that is composed of elastic links is referred to as Compliant Mechanism [1]. The absence of joints in Compliant mechanism enables the transfer of the input applied force through the components due to its elastic deformation [2], leading to the displacement of its output point [3]. Distributed Compliant Mechanisms have strain energy spread through the design, instead of being limited to specific components [4]. The less number of components makes these mechanisms to be quickly prototyped using three-dimensional printer. These mechanisms can be created using single material that is ideal for injection molding [5]. The effect of minimum number of components is higher efficiency and lesser friction compared to that of rigid bodies [6]. All these factors result in the decrease in inventory expenses for designing compliant mechanisms [7,8]. The Compliant mechanisms are ideal for small deformations and achieve high accuracy in micro-displacements [4]. The above factors also have demerits for various reasons. Since the individual elastic components can have varying deforming behavior, the Compliant mechanisms can be hard for design and analysis [9]. Unlike rigid mechanism, the analysis of force cannot be isolated from displacement for Compliant Mechanisms [4]. Allred TM also mentioned that this varying deformation can lead to wastage of internal energy, due to its higher distribution at undesired components. Compared to the rigid links, the elastic links in compliant mechanisms can have higher fatigue [9].

The practical applications of compliant mechanisms were developed by various researchers in the past. Frecker et al. developed a multifunctional compliant design for preventing tissue damage during surgeries involving indirect visualization [10]. A parallel kinematics inspired compliant mechanism was used to design a microwrist for needlescopy [11]. The design synthesis of compliant MEMS that carried out stroke amplification mechanism, was formulated using linear elastic models [1]. By altering the stiffness in flexural hinges, a decreased torque was obtained for flapping compliant wings [12]. A similar application generated a significant wing stroke through the deformation of a compliant mechanism in a micro aerial vehicle [13]. The parallel kinematic XY flexure mechanism was used in designing precision devices [40]. Compliant MEMS crash

sensors, comprising of snap-through buckling arcs, were analyzed using nonlinear Elastica theory [14]. Krishnan et al. developed displacement amplifying compliant mechanisms for the design of high accuracy microsensors [15]. Load path method had been used for design development of a human kidney manipulator, which is utilized as a device for performing surgeries involving indirect visualization [3]. Since there are more than one output point in shape morphing compliant mechanisms, the load path method can be used for designing various consumer products [16].

Various design methods for Planar Compliant Mechanisms (PCM) include Pseudo-Rigid Body Model (PRBM); Topology Optimization; Load Flow Visualization. PRBM was developed from an observation of a cantilever beam that underwent circular bending with its end point having large deflection. Some point on the undeflected portion turned out to be the center of the beam curvature. Hence, PRBM was used to predict the beam's end point deflection [6]. Howell et al. defined the pseudo rigid body angle (angle between the deflected and undeflected portion of the beam) in a cantilever beam replaced by 1-revolute joint PRBM [7]. The PRBM model used in [7] was enhanced through better PRBM parameter values [17]. Dado et al. enhanced the prediction accuracy of beam deflection through variable parametric 1-revolute PRBM [18]. The 2-revolute PRBM has been shown to be undergoing better end point deflection compared to that of 1-revolute PRBM [19]. Vitellaro et al. proposed an out-of-plane displacing MEMS actuator using PRBM method for design modeling [20].

Using the given parameters such as forces acting on input and output points lying on a design domain, the topology of a compliant design can be optimized [21]. One of the key finding of Sigmund's research was that the maximum stress can be kept under check through the addition of an input constraint. A Compliant transmission design was developed using Topology Optimization, in order to effectively combine it with electrostatic actuators that are used in microsystems [22]. Aguirre et al. developed a multifunctional compliant mechanism that had variable cross-sectional area through shape optimization [23]. This device is used as a surgical tool that reduces the number of tools needed to perform a surgery. Endoscopic surgeries require surgical tools that occupy limited space and this condition can be met by Compliant mechanism [24]. Using shape optimization, Cronin et al. refined a compliant mechanism that meets the design requirements of endoscopic surgical tool. The design of stroke amplification mechanisms was developed using energy efficiency formulation of topology optimization method [25]. Structural

optimization was performed for design synthesis of morphing aircraft structures to decrease the actuator force needed to produce desired deformation [26].

1.2 MOTIVATION AND BACKGROUND

The product design process generally involves creative designing followed by its refinement through simulation software. This process is painstaking because the designer needs to repeat this cycle till the desired design specifications are met. Though Topology optimization has been highly preferred method [27, 28] for designing compliant mechanisms because of its well-developed mathematical formulations, the simulations needed for the design refinement is computationally intensive. The fundamental issue in topology optimization is that topology and shape variables are assigned as parameters to the design domain. The topological solutions resulting from this optimization method are comprising of infeasible compliant structures as well [29, 30, 31]. As mentioned earlier, since PRBM is mainly focused on the end point deflection, this design method can get more complex when it is applied on distributed compliant mechanism [32, 4], wherein the stresses are not narrowed down to specific design component. A multi-port mechanism comprises of a minimum of two ports namely input and output points. The instant center of rotation provided the geometrical relationship between various components of a simple multi-port (one input and one output) mechanism [38]. The instant center approach was followed by the building block method [32], wherein the compliant mechanism is broken down into separate building block. The kinematics of each building block is assessed and then the cumulative kinematics of all the building blocks is the kinematics of the whole compliant mechanism. The key drawback of this research finding was that the compliance of the building blocks was not considered. The subsequent research of load flow visualization using kinetostatic formulation, wherein the building blocks were assessed both kinematically and elastically [33]. Thus, the load flow visualization method was preferred for the design synthesis of compliant mechanisms, mainly involving distributed compliance. The background of this thesis is focused on the Load Flow Visualization method for PCM because its extension to Spatial Compliant Mechanisms (SCM) using Virtual Reality forms the crux of this thesis research.

Load Flow Visualization involves the breakdown of a PCM/SCM design into individual components known as Load-Transmitter Constraint (LTC) sets. These components can be isolated and analyzed individually. This enables to treat a LTC set like a rigid body in a Free-body diagram. Then, the transferred force can be assessed for an individual LTC set and it can be applied to the subsequent LTC set. This topic will be covered in further detail in Chapter 3.

There have been few design guidelines using load flow visualization that had been established for PCM and mechanical metamaterials and they can be drawn using just pen and paper [34,35]. The three-dimensional geometry is more complex than that of the two-dimensional. It is cumbersome to complete the 3D design modeling without a proper visualization. Load flow visualization method reduces this complexity in modeling, but this method needs a visual medium for conceptual synthesis. In order to perform qualitative analysis of SCM through load flow visualization, there is a necessity for visual intuition. Virtual Reality not only supports visual intuition but also provides an immersive experience to the designer. It also catalyzes in rapid design modeling and qualitative analysis to validate SCM deformations, without the need for multiple iterations. This research is a novel attempt in applying load flow visualization method for conceptual synthesis of SCM using VR. The deciding factor for design feasibility is to detect the intersection/intermediate point but this would not be possible without proper visualization of the SCM. In addition, the intersected/truncated hemisphere at the intersection point is hard to visualize and eventually making it difficult in drawing a freedom line and constraint plane. Even the input/output hemispheres and other three-dimensional geometries are hard to visualize without VR. In 2D, for a freedom line, there is a corresponding constraint line, but in 3D, there is a corresponding constraint plane centered about that intermediate point. The constraint plane becomes hard to visualize, design and orient on a 2D paper. Hence, there is a need for Virtual Reality (VR) software platform for proper design visualization and provide different perspectives. The motivation of this thesis is to create a Virtual Reality (VR) software platform, enabling the designers to accelerate the design modeling and preliminary qualitative analysis of SCM that undergo small deformations, for meeting conceptual design specifications.

1.3 SCOPE

A three-dimensional VR design platform has been established for designing SCM using load flow visualization method in this thesis research. The other design methodologies such as PRBM and Topology Optimization for SCM are out of the scope of this thesis. There were no attempts made to apply these design methods on the VR design platform.

1.4 GOALS

The objective of this study is to develop a VR design platform that enables proper 3D visualization of some of the design guidelines that were hard to implement on pen and paper. The research aims include:

1. Development of architecture for creating essential features of the design software.
2. Development of VR software algorithm through the application of Unity3D [43] assets.
3. Testing and Validation of the software by checking if the 2D design guidelines can be implemented for modeling SCM. All the unfit 2D design guidelines of PCM will be replaced with corresponding 3D design guidelines, after modeling and analyzing the SCM on the VR software platform.
4. Enabling the user to move around this VR environment and getting an immersive experience of design modelling and analysis. Ensuring design domain can be scaled up/down in all directions and changing its orientation in any direction. This is a very important feature because scaling enables the user to analyze the deformation of a SCM design in a different view/perspective.
5. Performing trials of simple SCM Design modeling on the VR software.
    a) Check their feasibility instantly; make necessary changes to the design and ultimately obtain multiple SCM design derivatives.
    b) Eliminate the need for heavy computations for deriving and analyzing different modifications of a specific SCM.

1.5 THESIS OVERVIEW

The motivation of this thesis is to create a Virtual Reality (VR) software platform enabling the designers to accelerate the modeling and synthesis of SCM for meeting desired design

specifications. A 3-Dimensional VR design platform has been established for designing SCM using load flow visualization method in this thesis research. In order to accomplish this goal, the approach can divided into five stages: Development of architecture for creating essential features of the design software; Development of VR software algorithm through the application of Unity3D [43] assets; Testing and Validation of the software by checking if the 2D design guidelines can be implemented for modeling SCM; Enabling the user to move around this VR environment and get an immersive experience of design modelling and analysis; Performing trials of simple SCM Design modeling on the VR software.

Chapter 2 covers the key components/features of the VR design software; Properties of these key components; Relationship between these key components. The description of key components and their operation enables the designers to efficiently and effortlessly design SCM on this VR software. The properties and relationship between these key components are mentioned so that the user can customize the VR software according to their research needs. For instance, the user can adjust the lighting of the virtual environment, based on the properties detailed in this Chapter, if the existing luminosity is not optimum for viewing a specific part of SCM.

Chapter 3 provides literature review of load flow visualization methods for PCM. The basic building block known as LTC set is introduced and its application to various PCM are presented. The step-by-step guidelines for the design synthesis of various PCM is also covered.

Chapter 4 comprises of different SCM designs tested and validated on the VR software. The visual representation of deflected designs are compared on both the platforms namely Unity [43] VR and Matlab [51]. These comparisons clearly prove that VR software is a better application for visual representation of SCM and their deflections.

Chapter 5 concludes the thesis by summarizing the highlights of VR software and its application on load flow visualization of SCM. It also encapsulates the accomplishments of the current research and suggestions for future work such as feature extensions and applications of VR software.

CHAPTER 2: VR DESIGN SOFTWARE

2.1 INTRODUCTION TO THE VR DESIGN SOFTWARE

The interface of VR Design software mainly comprises of the virtual environment and design domain. The virtual environment is an enclosed room and it has a spot light fixed at the center of the room (Figure 2.1). The user can move around this environment and view the Compliant Mechanism design from different perspectives. The intensity of spot light ensures that the brightness of the design domain is higher than the surrounding virtual environment. This feature highlights the design domain, so that the user can focus attention only on the compliant mechanism design.

This VR design software has been developed on Unity game engine [43] Version 5.6.2f1 and the scripting has been performed on the C# Programming language. The key design Unity [43] assets that were used for software development include VRTK (Virtual Reality Toolkit) [44]; Vectrosity [45]. The hardware compatible to this VR software is Oculus Rift with Touch Controllers [46].

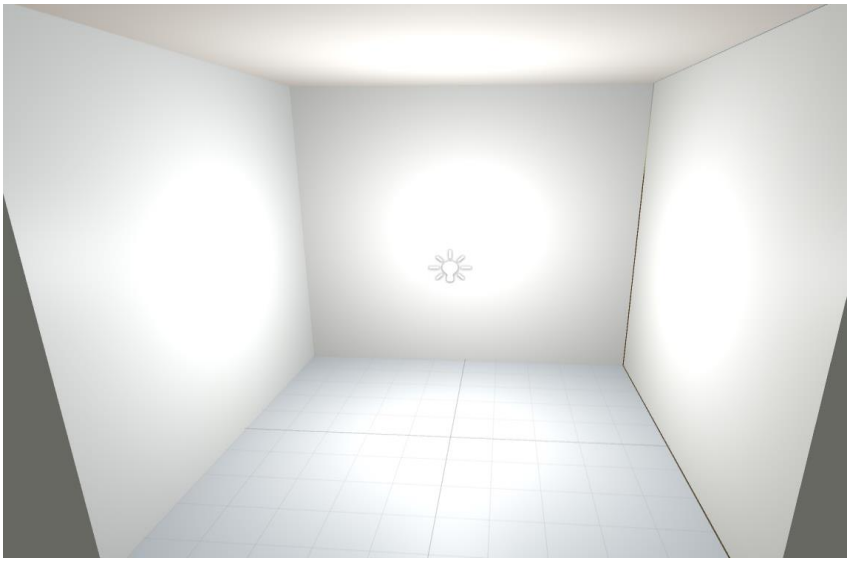


Figure 2.1: User Environment

2.2 KEY COMPONENTS IN VR DESIGN SOFTWARE

The radial menu comprises of various buttons to select and click for the activation of corresponding tools. In this VR software, the radial menu is called Compliant Tools menu (Figure 2.2) comprising of all the essential design components such as input point, output point, intermediate point, fixed point, cancel tool, force vector, freedom ray.

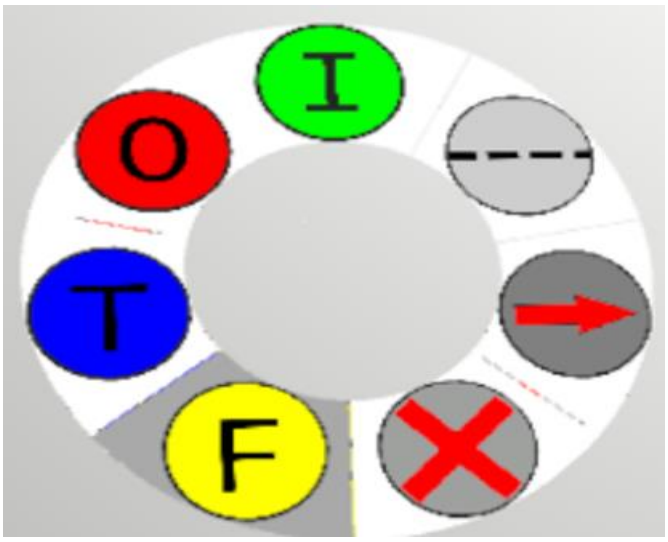


Figure 2.2: Compliant tools menu comprising of key components of the software

The key components of VR design software include:

2.2.1 DESIGN DOMAIN

It is a 3D space in which the compliant mechanism designs are drawn free hand by the user. The design domain is in the shape of a cube and it is placed at the center of the room by default. The design domain also enables boundary constraints (covered in Chapters 3 and 4) to be placed on its faces.

In addition, the cube is transparent so that the intermediate points are visible while design modelling. As it can be observed in the Figure 2.3, the coordinate axes X, Y, Z are denoted by red, green and blue arrows respectively; while the origin is denoted by a small white sphere.

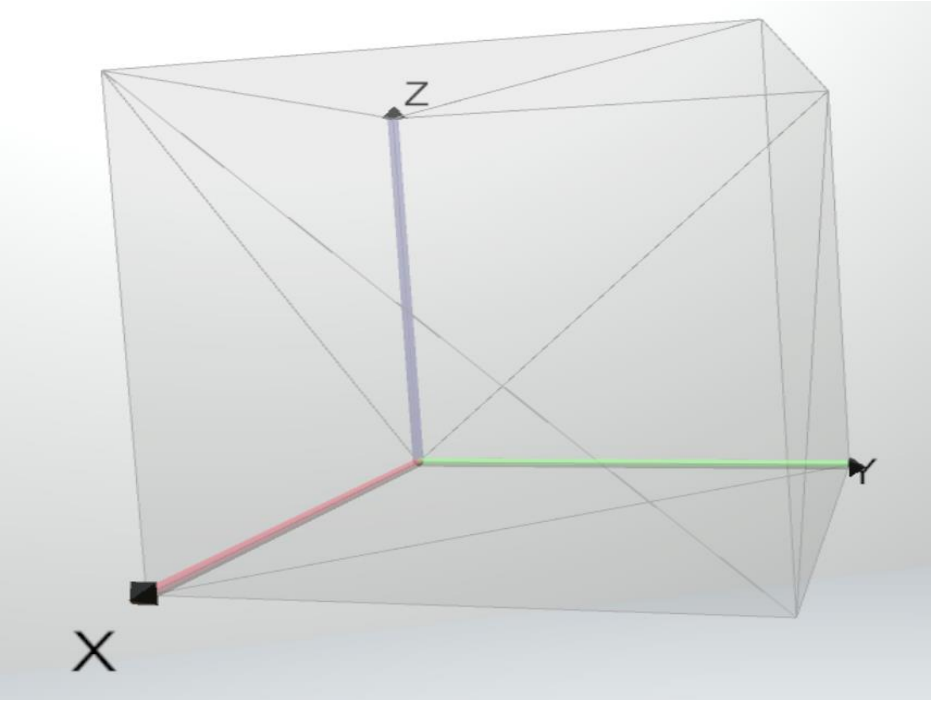
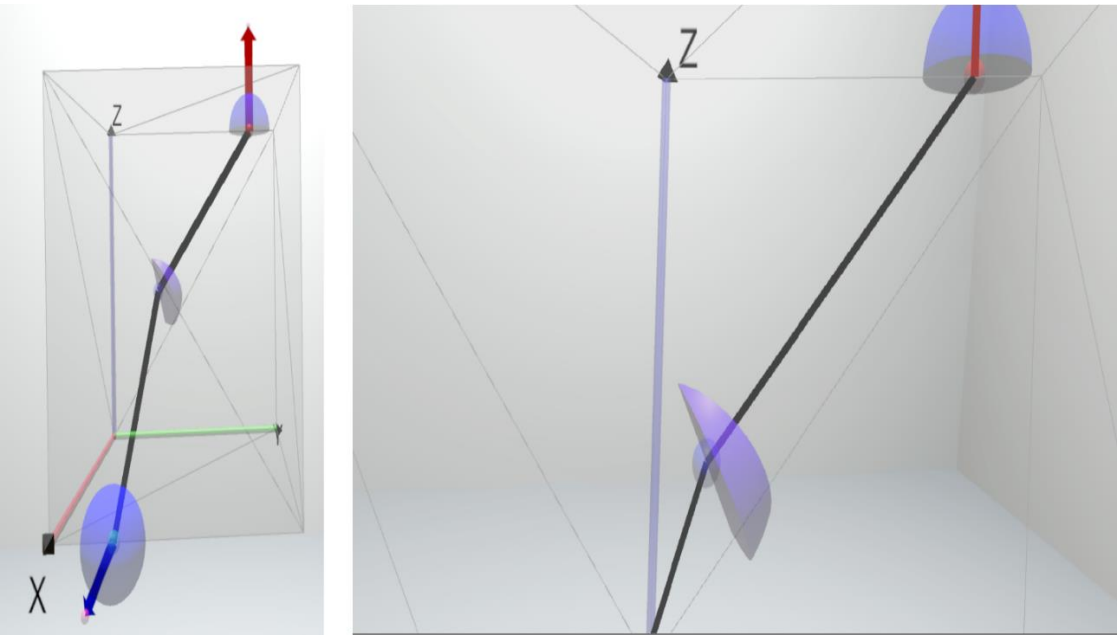


Figure 2.3: Design domain (Cube) with coordinate axes

The user can grab and hold the design domain (cube) for any amount of time. Eventually, the user can drop the cube at a desired location of the room. The cube can also be scaled up/down in all directions and its orientation can be changed in any direction (Figure 2.4). This is a very important feature because scaling enables the user to analyze the deformation of a compliant mechanism design in a deeper perspective. Also, the orientation enables the user to observe the compliant design from different view angles. The only drawback is that the orientation does not work when the design domain is scaled up to the user's maximum arms abduction. During this situation, the user can physically move around in real world and camera sensor in Oculus Rift [46] follows the user movement. This makes the user feel as if moving around the design domain and the user can view the design from different perspectives.



(a) (b)

Figure 2.4: Design Domain a) before scaling b) after scaling (user is inside the design domain)

2.2.2 INPUT, OUTPUT, INTERMEDIATE AND FIXED POINTS

The Input, Output and Intermediate points are the key nodes of the compliant mechanism design. They define the location of constraints placed on the compliant design with respect to the location of fixed points placed on design domain. The output point undergoes displacement based on the magnitude and direction of the force applied on the input point [34, 35]. Correspondingly, the input, intermediate and output points get deflected depending on the force magnitude and direction. These points are represented by spheres in the VR software but they do not undergo any spherical deformations due to the applied forces. It is only meant for a clear visual representation of these points, since there are generally small deflections in compliant mechanisms. The color representation of the point spheres are Green (Input); Red (Output); Yellow (Fixed) and Blue (Intermediate) (Figure 2.5).

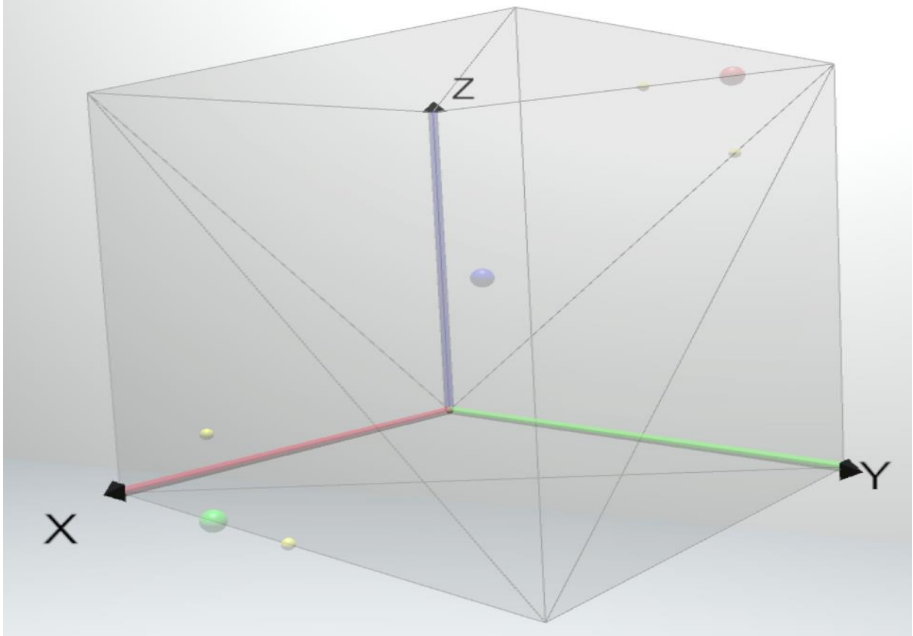
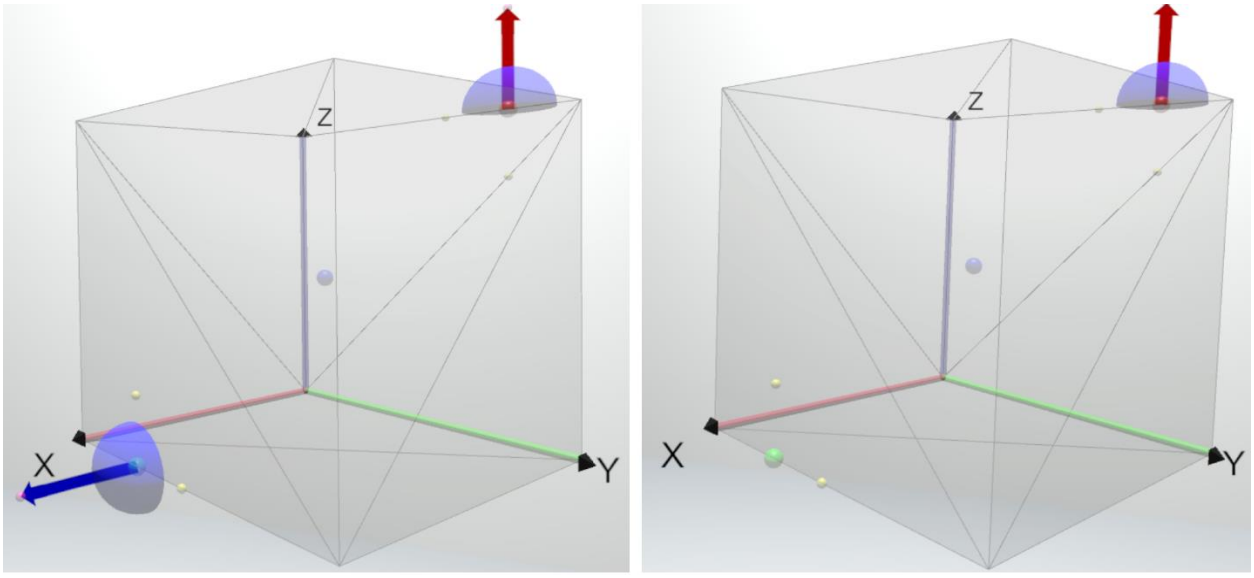


Figure 2.5: Input (Green), Intermediate (Blue), Output (Red) and Fixed (Yellow) Point Spheres on design domain

2.2.3 CANCEL TOOL

If the user would prefer to change the location of any points (input, output, fixed, intermediate) or they created an unwanted point/line, then these features can be removed by choosing cancel tool. It is also used to undo a previous operation. For instance, if the user is not satisfied with the freedom line drawn, it can be removed, and the software returns to the previous step of drawing freedom line. The Cancel tool treats a line and a force vector in a similar fashion. Hence, when a point attached to a force vector is removed, the force vector is also eliminated. If only the force vector needs to be eliminated, then the point sphere on the head end of the vector needs to be removed (pink sphere in Figure 2.6). This ensures that the input/output point are not removed during this operation. The drawback of this tool is that it cannot skip multiple previous steps performed on the design domain.



(a) (b)

Figure 2.6: a) Force Vector on Input Point present before using Cancel tool b) Force Vector on Input point removed after applying Cancel tool

It is denoted by a red cross symbol on the left touch controller [46]. The user can navigate to the 'X' button on the tools menu using the left joystick and press it to select the cancel tool (Figure 2.7). It can also be used to delete any force vector on various points on the design. The algorithm ensures effortless removal of points/lines, since there are dim grey colored points at the end of every line. So, the user needs to get close to the desired point and then it would get highlighted. On pressing Y button on the left joystick, the point gets removed and any corresponding line attached to that point will also be removed (Figure 2.7).

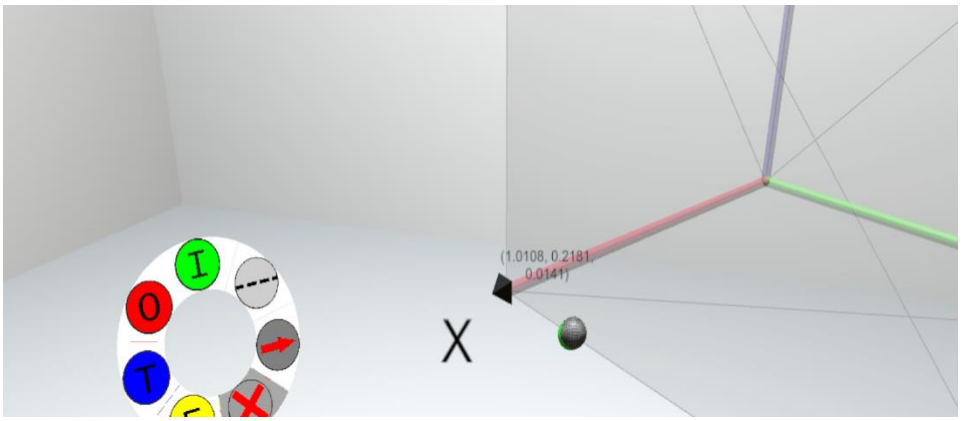


Figure 2.7: Using Cancel Tool (X Symbol on Compliant tool menu) to delete input point by placing cancel sphere (black) over input point sphere (green) using right controller [46]

2.2.4 FORCE ACTING ON INPUT AND OUTPUT POINTS

The direction of input force vector is crucial for the behavior of the compliant mechanism (Figure 2.8). If the user already has the exact magnitude of force vector in X, Y, Z directions, then the force vector can be drawn by free hand and keep parallely observing the force magnitude values of real-time text. Once the desired magnitude is reached, the user can stop the drawing motion and the force vector gets locked at that specific magnitude. The real-time text continuously keeps track of the force vector coordinates every frame and depicts its values spontaneously. An important point to note is that the force vector is automatically disabled when it is being drawn on any other points apart from input and output points. It is also disabled on any existing lines and this disable feature avoids any accidental placement of the force vector.

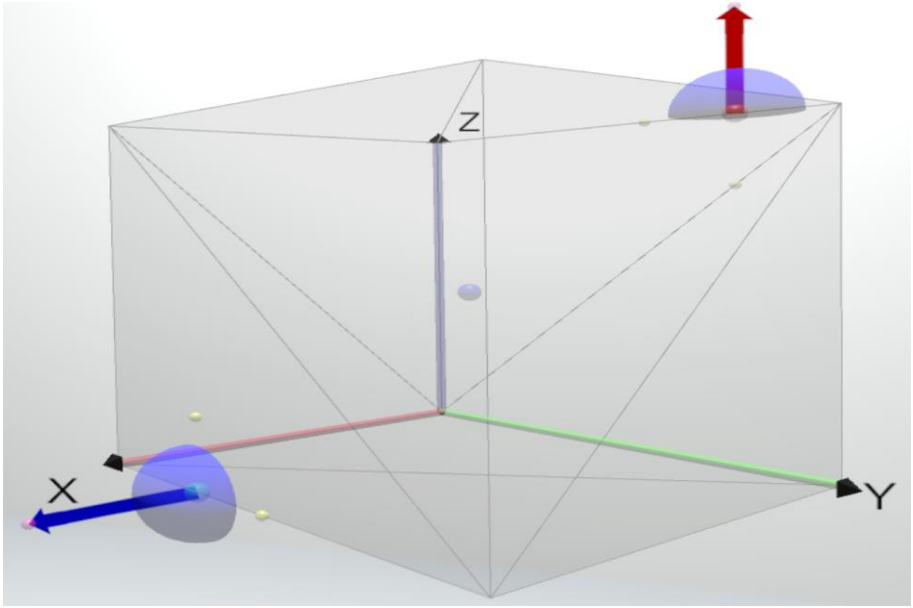


Figure 2.8: Force vectors at input (Blue arrow) and output point (Red arrow) spheres

2.2.5 TRANSMITTERS AND HEMISPHERICAL BANDS

Transmitters are lines drawn between input, intermediate and output points and they depict load flow direction (Figure 2.9). Hemispherical bands indicate the possible load flow directions and hence related to load path in the transmitters, based on the input and output points (Figure 2.9) [34, 35]. The base of a modified hemispherical band is perpendicular to the transmitter attached to it. The size of these truncated bands varies with the change in relative position of the output point with respect to the input point.

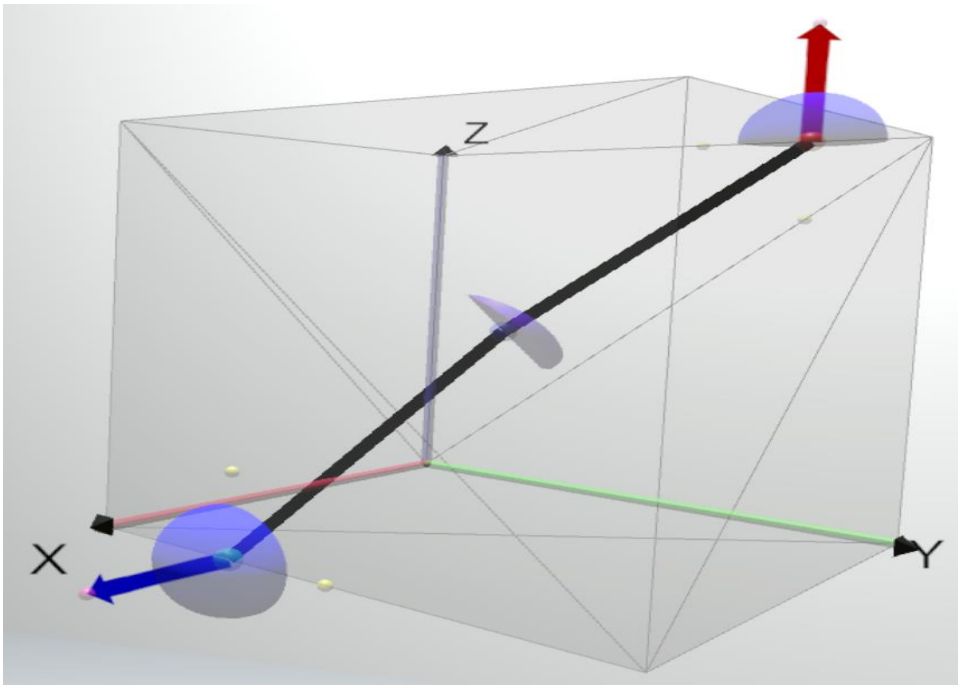


Figure 2.9: (i) Transmitters (Black Lines) between input-intermediate-output point spheres and (ii) Hemispheres (Blue) at input, intermediate (truncated) and output point spheres

2.2.6 FREEDOM LINE AND CONSTRAINT PLANE

The Freedom line and Constraint plane are applied on the intersection point i.e. at the bifurcation of the load path between input and output points (Figure 2.10). The freedom line is a direction restricted with a range of directions possible in the intersected hemispherical band at the intersection point (intersection between two transmitters) [34, 35]. It is the initial step in deciding in which direction the compliant mechanism moves.

In 2D Compliant mechanism, a single freedom line generated a single corresponding constraint line [34, 35]. On extending this principle, a single freedom 3D ray generated a corresponding constraint plane and hence two constraint lines are needed. The constraint lines need to be both perpendicular to the freedom line. The direction of these constraint lines plays a pivotal role in the axial load flow within these transmitters.

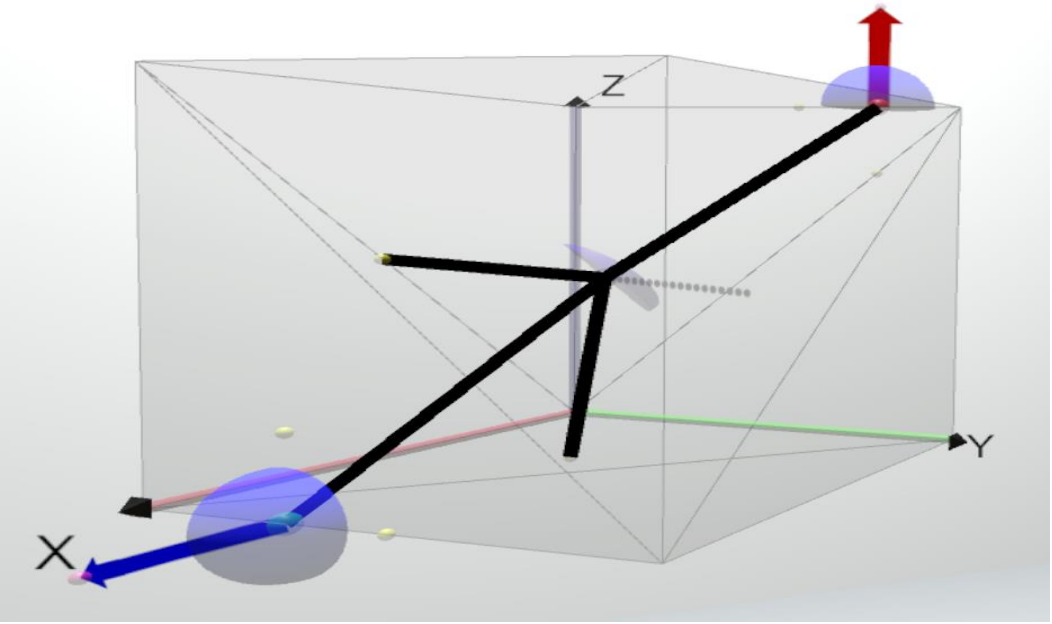
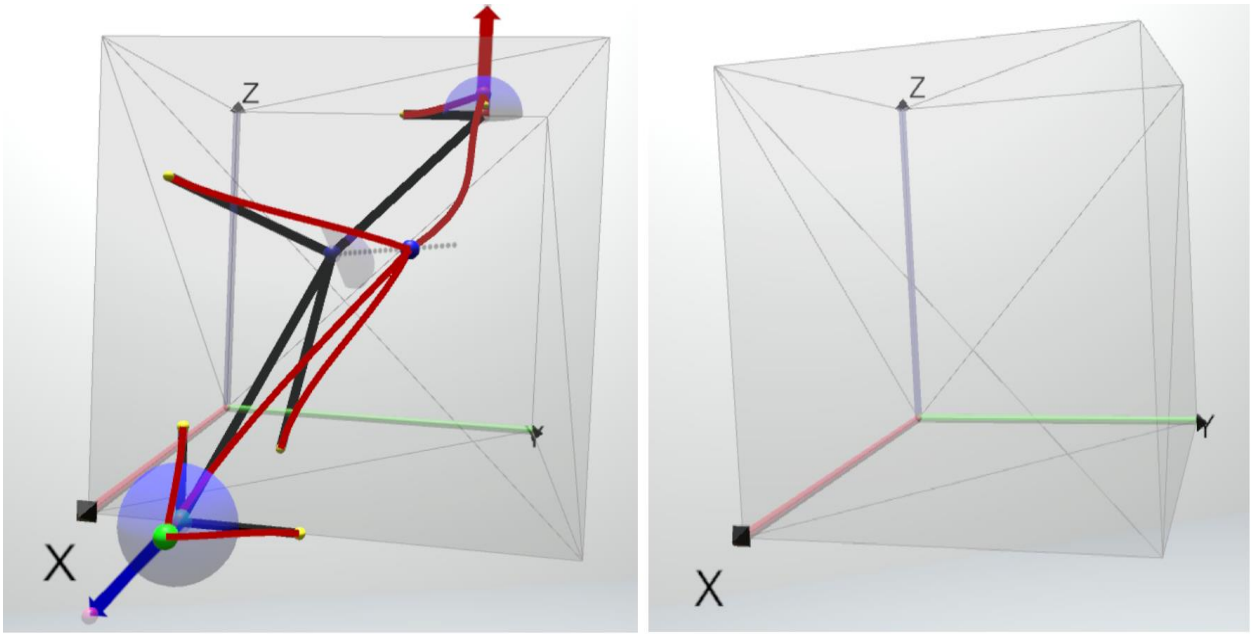


Figure 2.10: Freedom line (Dotted Line) and Constraint Lines (thick black line connecting blue intermediate and yellow fixed point spheres) connected to intermediate point sphere

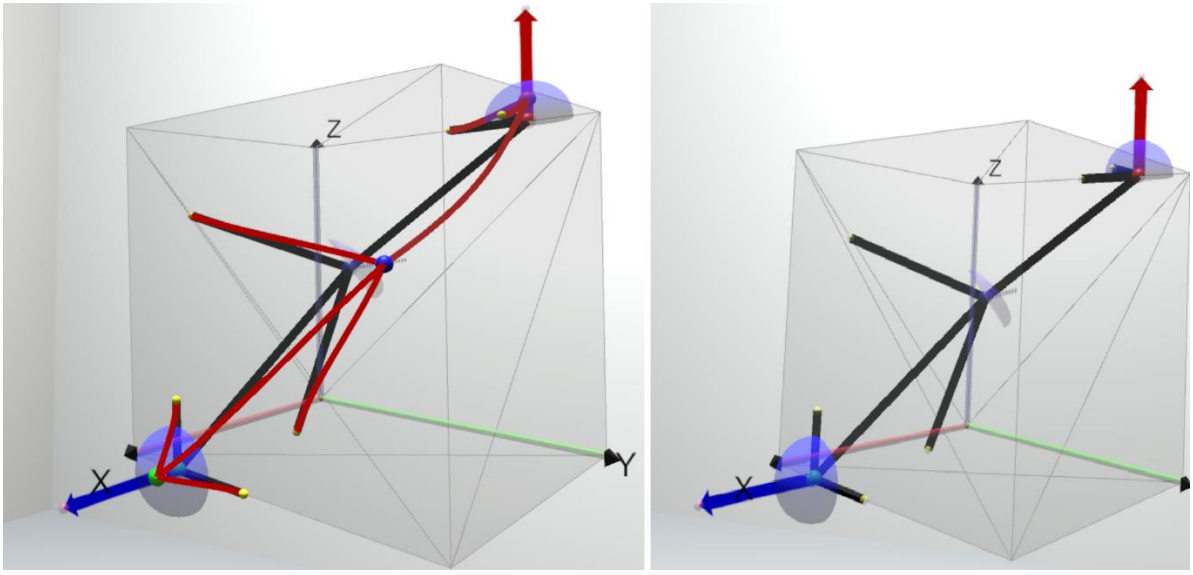
2.2.7 RESTART DESIGN AND RESET ANALYSIS TOOLS

Restart Design tool is used when the compliant design turns out to be infeasible in all possible cases and there is a need to restart with a new compliant design (Figure 2.11). Reset Analysis tool lets the user get back to the design modelling phase, if they feel that a specific design element such as freedom line, constraint lines etc. are not providing feasible result after design analysis (Figure 2.12). Hence, this tool lets the user return to the previous state of submitting the design for analysis. From there, the cancel tool can be utilized for undoing each step of design execution.



(a) (b)

Figure 2.11: Design Domain a) before using Restart design tool b) after using Restart design tool



(a) (b)

Figure 2.12: Design Domain a) before using Reset Analysis tool b) after using Reset Analysis tool

After analyzing the deflected design, if the user feels dissatisfied with the deflected design, then the user can make changes to the original design by pressing button B on the right controller [46]. Then, a green colored ray cast is released from the right virtual hand on to the "Restart" button on one of the walls of the room. This button will be highlighted in yellow color, once the ray cast is concentrated on it and then the fire button on right controller [46] needs to be pressed, in order to press the Restart button. Then the design domain (cube) becomes empty and the user can draw a new compliant design and repeat the above design process (Figure 2.13).

Figure 2.13: Shooting green raycast from Oculus Touch Controller [46] on to the Restart button

A similar operation to that of Restart button is performed on Reset Analysis button but the only difference is that the raycast will be targeted on Reset Analysis button. After pressing the button, the design domain will remove all the red lines and keep the black lines, so that the user can make changes to the existing compliant design before performing new analysis on it (Figure 2.14).

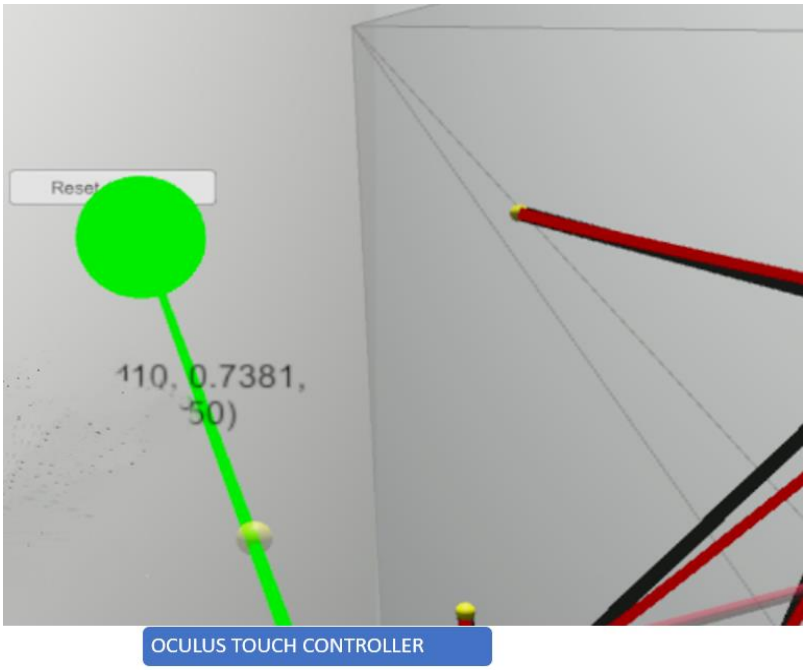


Figure 2.14: Shooting green raycast from Oculus Touch Controller [46] on to Reset Analysis button

2.3 PROPERTIES OF KEY COMPONENTS IN VR DESIGN SOFTWARE

2.3.1 PROPERTIES OF DESIGN DOMAIN

2.3.1.1 UNITY EDITOR INTERFACE

The Transform component defines the local position, rotation and scale properties of the design domain in the virtual environment [48]. Each property comprise of X, Y and Z sub-components and it enables change in a specific direction. The Transform component is a default component in all the gameobjects. The Mesh Filter component applies the Cube Mesh i.e. quadrangulated polygon mesh on to the Design Domain and hence rendered as a cube by Mesh Renderer component. The key thing to note in Mesh Renderer component is that the design domain receives shadows cast by other gameobjects. The Box Collider component ensures that the design domain interacts with other gameobjects and all kinds of physics principles can be applied on it. The RigidBody component includes various physics principles such as Drag, Gravity, Kinematics and RigidBody Collision. The Kinematics is enabled for the design domain so that it can be mobile,

and its motion is controlled by the user. The Gravity is not enabled since it will fall off to the ground otherwise, instead of floating in space. The Resize Object Script component is used to scale up/down the design domain using the Oculus Touch Controllers [46] (Figure 2.15).

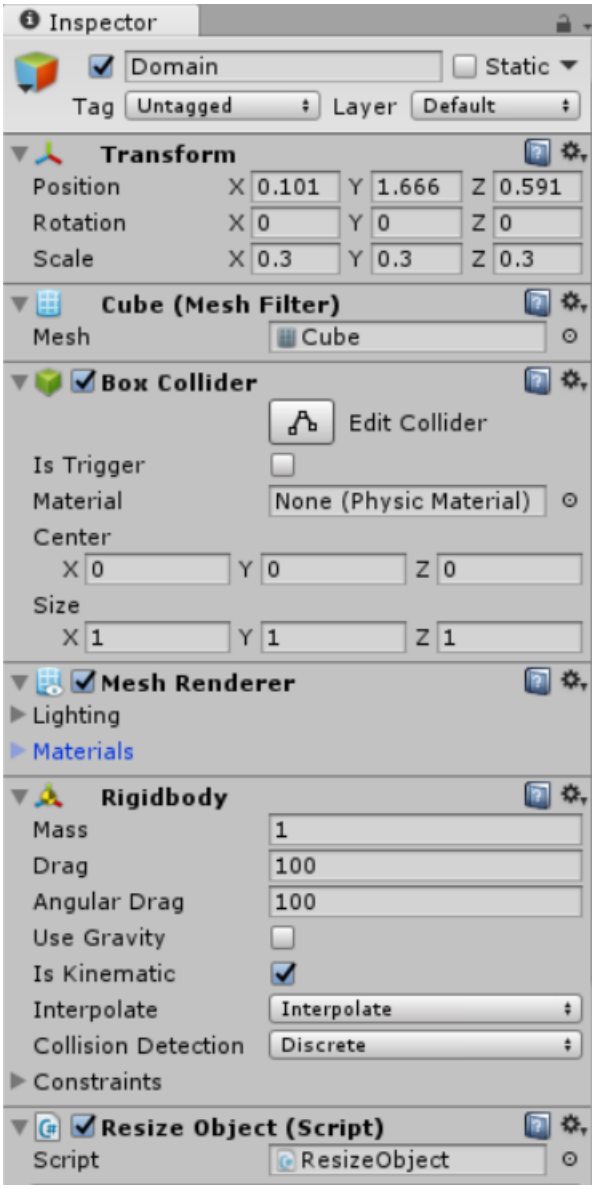


Figure 2.15: Unity Editor [43] showing the properties of design domain (With the permission of Unity Technologies. All rights reserved.)

VRTK_Interactable Object Script component ensures that the design domain can be grabbed using the Oculus Touch Controllers [46]. In this script, the key properties include grab idle state, pressed controller [46] button, grab attachment behavior and secondary grab action. The grab idle state property makes sure that this script is active only when the design domain is grabbed by the user [44]. The pressed controller [46] button refers to a specific button on the controller [46] that activates the grabbing action on design domain [44]. The attachment/clinging style of the design domain on to the Touch Controllers [46] is determined by grab attachment behavior property [44]. If the user uses right hand simultaneously when left hand is used for grabbing the design domain, then the design domain switches to the right controller [46] because of secondary grab action property [44].

VRTK_Fixed Joint Grab Attach Script component creates a connection joint between the design domain and the Oculus Touch Controller [46, 44]. The key parameter is Break Force and it is set to infinity. This means that the connection joint is impossible to break and get detached. VRTK_Swap Controller Grab Action Script gameobject lets the user shift the gameobject from left hand to right hand and vice versa by creating a connection joint in the later hand and break the connection in the former hand [44]. Init Lines Script component lets the user to draw the transmitters between various points such as Input, Output and Intermediate. The Camera is attached to this component so that the camera is placed right in front of the line, hence it is easy for the user to follow the line. Finally, the Shader component defines the material properties of the design domain [48]. It is a standard silver colored material. The design domain is transparent because of the Rendering Mode property. It also has a glass kind of reflective property since Reflections is enabled.

2.3.1.2 COORDINATES OF THE DOMAIN

The coordinates of the input point, shown above the green input point sphere (Figure 2.16), gets updated in real time, as the user moves the input point around different parts of cube, using the right Oculus Touch Controller [46]. This is a very useful feature since it helps the user to place a desired point in the exact location, if the user knows the coordinates prior to the modelling.

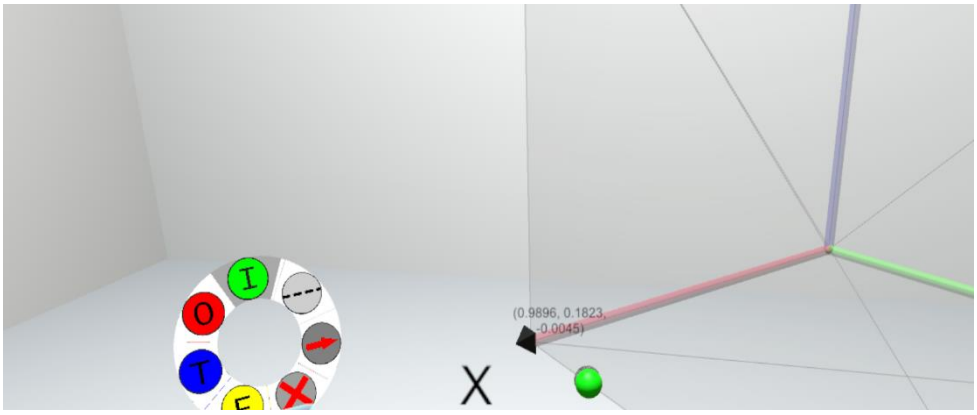


Figure 2.16: Coordinates of input point (green sphere) on the design domain

The relevant script can be found in Appendix A and its main function is to capture the coordinates of various points on the design domain. The local coordinates of the cube is set with respect to the origin of the domain space. It is obtained by converting the global position of the preview gameobject into the local position with respect to the domain. The preview Gameobject helps the user to visually see the coordinates (X, Y, Z) of cursor, as the user moves the cursor over the volume of the cube domain. This is a very useful feature when the user knows the coordinates of a specific point and desires to place it on the domain. These coordinates are transferred to a Textbox over the right controller [46] and it gets updated in real time, as the user moves the cursor around the domain space.

2.3.1.3 RESIZE OF THE DOMAIN

The relevant script can be found in Appendix A and its main function is to scaling up/down the design domain. When the squeeze triggers of both left and right Oculus Touch controllers [46] are pressed, the software checks if the distance between the left and right hand of the user has increased or decreased. If this turns out to be true, then the software scales up/down the size of design domain. Finally, the cube domain is locked down to the scaled magnitude set by the user, once they release one or both of the squeeze triggers.

2.3.2 PROPERTIES OF INPUT, OUTPUT AND INTERMEDIATE POINTS

2.3.2.1 DATA EXTRACTION OF INPUT AND OUTPUT POINTS

The relevant script can be found in Appendix A and its main function is to collect the data of all the key components that are required for the design analysis in Matlab [51]. These components only pertain to input and output points. The key components include force vector (magnitude and direction) and hemisphere. The data values that are transferred to Matlab [51] include position coordinates of tail and head end of the force vector; direction of force vector; hemisphere; add and remove new and old points respectively to the list of points in a transmitter.

2.3.2.2 DATA EXTRACTION OF INTERMEDIATE POINT

The relevant script can be found in Appendix A and its main function is to collect the data of all the key components that are required for the design analysis in Matlab [51]. These components only pertain to intermediate points. The data values that are transferred to Matlab [51] include position coordinates of intermediate point; points on the transmitters intersecting at intermediate point; truncated hemisphere; freedom line; constraint plane.

2.3.2.3 COLOR FOR INTERMEDIATE POINT AND TRUNCATED HEMISPHERE

The relevant script can be found in Appendix A and its main function is to apply same color to the intermediate point and truncated hemisphere. This script ensures that the intermediate point and truncated hemisphere are highlighted with zero transparency, when the mouse cursor is placed on them. Otherwise, they remain with a partial transparency. The same color is ensured for both these components by matching their material properties.

2.3.3 PROPERTIES OF RADIAL MENU

VRTK_Radial Menu Script component pops up a radial menu over the Oculus Touch Controller [46, 44]. The menu comprises of various buttons to select and click for the activation of corresponding tools. In this VR software, the radial menu is called Compliant Tools menu

comprising of all the essential design components. The Button Prefab property sets the base of a specific button and the base has a material and color properties so that it appears as an opaque foundation [44]. The Generate on Awake property enables the radial menu to be visible as soon as the software is initialized [44]. Hide On Release property makes the radial menu invisible, when the user removes the thumb over the thumbstick of left controller [46, 44]. Execute on unclick property releases the pressed button and deactivates the corresponding component [44].

The Buttons property comprises of size, icon and On Click() event. The size refers to the number of interactive buttons to be placed on the radial menu [44]. In this VR software, there are a total of 7 buttons namely Input, Output, Intermediate, Fixed, Delete, Force and Freedom components. They are denoted by I, O, T, F, X, Arrow, Dotted Line respectively. The first element (Element 0) in this array is Input button and it is represented by green colored 'I' image on the radial menu. The On Click() event comprises of three key parameters namely event handling, Reference Script, Relevant Called Function, Index [44]. Event handling declares when to allow the button to be clicked, during editing time or Runtime or both [44]. The referenced script decides a specific action to be performed when the button is clicked. The called function within this referenced script has the exact operation the script needs to execute. The Index refers to the button number within this called function. Similarly, the second element (Element 1) in this array is Output button and it is represented by red colored 'O' image on the radial menu. The OnClick() event is the same as that of the Input button, except the index of button number 1. The other buttons have similar parameters applied with the index applied accordingly [44].

2.3.4 PROPERTIES OF FREEDOM LINE AND CONSTRAINT PLANE

The relevant script can be found in Appendix A and its main function is to create and destroy freedom line and constraint plane. It makes sure that the constraint plane is created as soon as the freedom line is drawn. It also makes sure that it gets destroyed as soon as the constraint lines are drawn. The Constraint plane is spawned perpendicular to the freedom line and it is always attached to the intermediate point. The plane also has a sensing collider that keeps track of the user's hand movements and the corresponding constraint line is drawn on it.

2.3.5 PROPERTIES OF SUBMIT DESIGN AND RESTART BUTTONS

2.3.5.1 UNITY EDITOR INTERFACE

These buttons are similar to that of the interactive buttons used in the VRTK [44] Radial Menu on the left controller [46]. The key difference is that these buttons are default in Unity [43]; while VRTK Radial Menu is part of VRTK asset [44]. It can be observed that the OnClick() event in the Button property are called only during the runtime for both the buttons. Also, the reference scripts are Networking (Figure 2.17) and ResetAll (Figure 2.18) for Submit Design and Restart buttons respectively. In the Networking script, the submit function is called, which sets up the socket between Unity [43] and Matlab [51] and transfers all the essential data (points and force vectors) from Unity [43] to Matlab [51]. In ResetAll script, the ResetAllButton function reloads the scene back to the default.

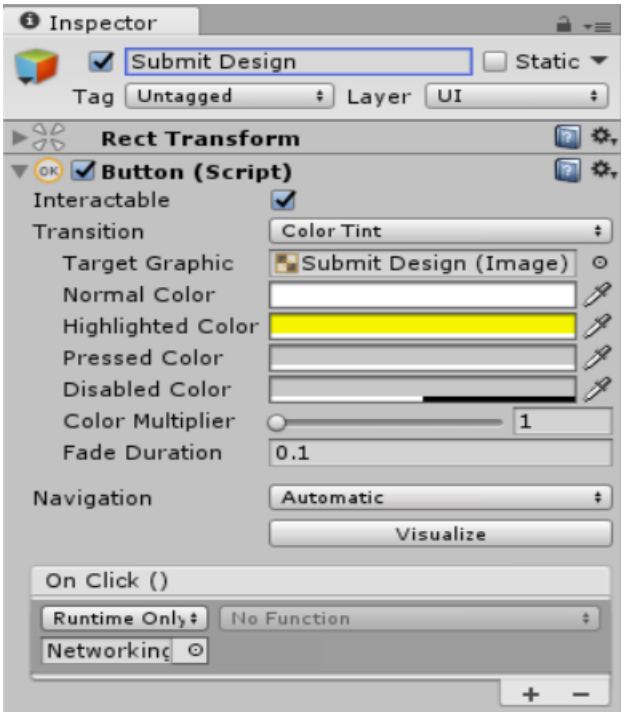


Figure 2.17: Unity Editor [43] showing properties of Submit Design button (With the permission of Unity Technologies. All rights reserved.)

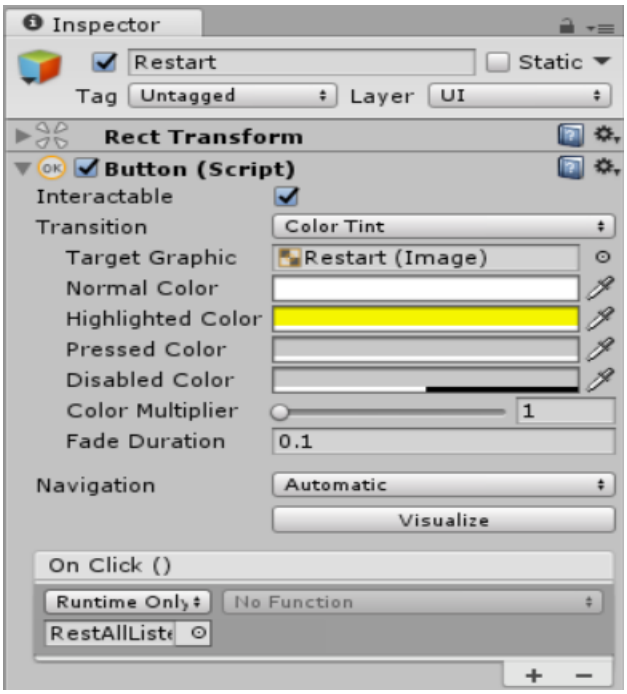


Figure 2.18: Unity Editor [43] showing properties of Restart button (With the permission of Unity Technologies. All rights reserved.)

2.3.5.2 SUBMIT DESIGN TOOL AND CONNECTION BETWEEN UNITY AND MATLAB

The relevant script can be found in Appendix A and its main function is to activate/deactivate 'submit design' button and also establish socket connection between Unity [43] and Matlab [51] by port number. The integration of Matlab [51] and Unity [43] requires important libraries such as System.Net and System.Net.Sockets [48]. The data receiver is activated first and it waits till the sender gets connected to it. When there is connection established, the activation of Submit Design button sends the data from Unity [43] to Matlab [51]. This script also constantly checks if there is a connection between Unity [43] and Matlab [51]. The connection is broken immediately after the Matlab [51] produces the results of design analysis.

2.3.6 PROPERTIES OF RESET ALL BUTTON

The relevant script can be found in Appendix A and its main function is to activate/deactivate 'reset all' button. In order to reload a specific Unity [43] scene, the key library

to be accessed is the UnityEngine.SceneManagement [48]. Initially, the SceneManager recovers the currently active scene by its name and that specific scene is loaded [48]. This eventually leads to reloading the game back to the beginning of empty domain.

2.3.7 PROPERTIES OF HEMISPHERE

2.3.7.1 CREATION AND TRUNCATION OF HEMISPHERES

The relevant script can be found in Appendix A and its main function is to create hemispheres at input and output points. It also generates truncated hemisphere (intersection of input and output hemisphere) at the intermediate point. The freedom line vector is activated for use immediately after the activation of truncated hemisphere. When the user hides the output hemisphere, the truncated hemisphere at the intermediate point is converted to the input hemisphere, since the connection is lost. As a consequence, the truncated hemisphere data gets removed from the list of data to be transferred to Matlab [51]. The truncated hemisphere will be recalculated once there are both input and output hemispheres.

2.3.7.2 TOGGLE VISIBILITY OF HEMISPHERES

The relevant script can be found in Appendix A and its main function is to toggle hemispheres at all points (input, intermediate and output) of the design. Initially, the hemispheres are made active and visible, since that is the desired outcome. This script also ensures that the freedom line vector is kept visible after it is being drawn by the user and continues to be visible even after the design analysis. The toggle action will be applied by pressing button ‘X’ on the left controller [46]. Hence, if the hemisphere was previously visible, then it can be made invisible by pressing button ‘X’.

2.3.8 PROPERTIES OF TRANSMITTERS [LINES BETWEEN THE POINTS]

The relevant script can be found in Appendix A and its main function is to draw any kind of line in the design domain. The different kinds of line comprise of the main lines and deformed

lines between the Input, Intermediate and Output points; freedom line; constraint line. The key asset used here is Vectrosity [45]. In addition, there are mesh lines drawn around the cube, thus forming a wireframe, which are used for defining the cube boundaries and enable its interaction with player controller [46]. The data of all the points corresponding to a specific line is transferred to Matlab [51]. In order to draw a line effectively, Vectrosity [45] needs a reference of the camera (CenterEyeAnchor Camera) being used.

2.4 RELATIONSHIP BETWEEN KEY COMPONENTS IN VR DESIGN SOFTWARE

In Unity3D [43] game engine, empty gameobjects are used to contain independent scripts that define relationship between key components.

2.4.1 SWITCH BETWEEN DIFFERENT TYPES OF POINTS

2.4.1.1 UNITY EDITOR INTERFACE

Point Type Switcher Script component handles switching between different types of point spheres. This is achieved through SwitchTo() function which is called whenever the VRTK [44] Radial Menu object triggers an OnClick() event on the left controller [46]. It shows the currently active point sphere on the right controller [46]. For instance, if the Input button is pressed on the radial menu, then it shows a green point sphere over the right controller [46], so that the user can place the Input point on the design domain. This script component comprises of various parameters such as Fix Axes Radial Menu, Force Radial Menu and Force Canvas. Fix Axes Radial Menu (Figure 2.19) and Force Radial Menu (Figure 2.20) lets the user lock specific axes (X, Y or Z) of a point (Input, Output, Intermediate, Fixed) and Force vector respectively, so that they can place that point at a precise location by free hand. Force Canvas is a real-time text box that shows the current magnitude of the Force vector in X, Y and Z directions. It gets updated every frame.

Spawn Object Script component applies to all the different types of point spheres on the right controller [46]. It controls how and when the spheres are allowed to be placed. If Allow Drag parameter is true, it allows the user to drag to create a line between two points. If it turns out to be false, then it will only place point at origin point. Allow Drag is true for both intermediate and

fixed points, while it is false for input and output points. Restrict To Boundary parameter determines whether a point sphere should be restricted to the boundary of a design domain or not. It is true for Input and Output points while it is false for intermediate and fixed. Restrict to Inside determines whether a point sphere should be restricted to inside the domain or not. It is true for intermediate point and false for all the rest of the point spheres. The Preview parameter refers to the preview sphere which is greyish white in color and depicts the preview of a point sphere which is going to be placed over the design domain (Figure 2.21).

Figure 2.19: Fix Axes Radial Menu for locking the X, Y or Z directions for point spheres

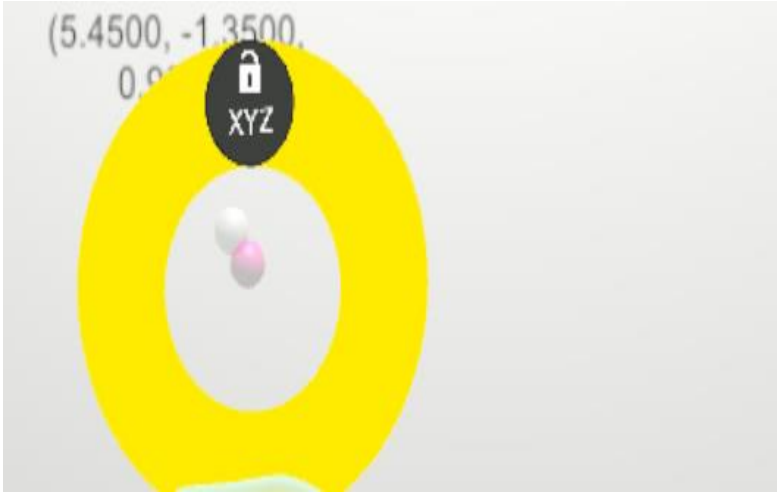


Figure 2.20: Force Radial Menu for locking X, Y or Z directions of force vector

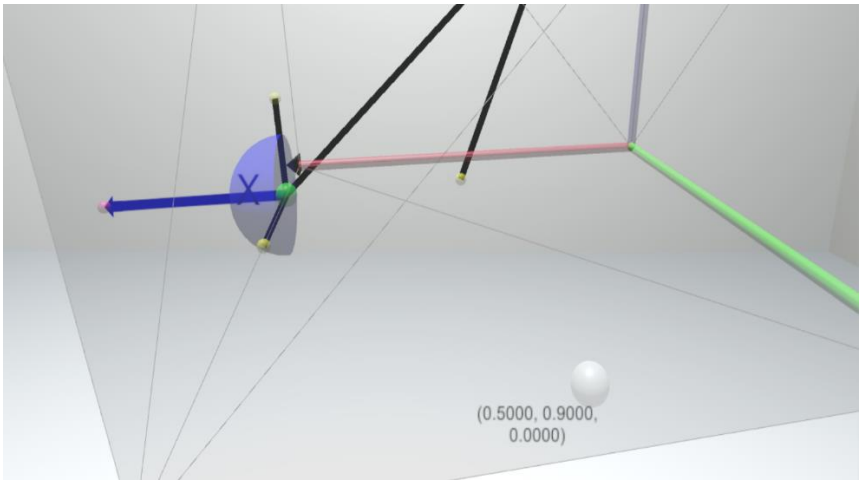


Figure 2.21: The preview point sphere is represented by the greyish-white sphere

2.4.1.2 POINT TYPE SWITCHER

The relevant script can be found in Appendix A and its main function is to transition between Input, Output, Intermediate, Fixed point spheres. It also involves few additional elements such as Freedom Line sphere, Force Vector Sphere, Cancel sphere. Though these additional elements are vector lines, they are represented by a sphere such that this sphere is used as a drawing tool (like a paint brush) to draw the respective vector line. This transition of points can be observed on the sphere just above the right controller [46]. For instance, in Figure 2.16, there is a green sphere above the right controller [46] and it represents that the input point is currently active.

It gets activated On Click event i.e. when the user places their thumb over the left joystick. The software behind this event handling User Interface mechanism is Virtual Reality toolkit. The menu template was already available, and it is customizable according to the needs of the user. The force Canvas is activated when the force button is pressed. Otherwise, it is deactivated so that it does not annoy the user with continuous updates of Force vector values in real-time, even when the Force option is not used.

CHAPTER 3: REVIEW OF LOAD FLOW VISUALIZATION

This Chapter is a brief review of the research findings of Krishnan et al on Load Flow Visualization method using Load Transmitter-Constraint sets [33] [36] [37], which forms the foundation for this thesis work. Krishnan et al demonstrated an efficient way of extracting the basic building block of a 2D compliant mechanism. This building block sets the foundation for load flow visualization in compliant mechanism and hence ensuring an effective design synthesis [37]. Prior to the contribution of Krishnan et al.'s research, the load flow visualization, which was part of theoretical design, involved an innovative design process and then it was validated using various optimization techniques. Due to derivation of a mathematical algorithm by Krishnan et al., the design synthesis could also involve load flow method [33]. Patiballa et al. later refined these design guidelines for both PCM as well as mechanical metamaterials [34, 35].

The key components of this Chapter include definition of Load Flow and LTC Sets; Properties of LTC Sets; Step-by step algorithm for design synthesis of planar compliant mechanism.

3.1 LOAD FLOW

The input applied force is transferred within the members of the compliant mechanism in the form of virtual force field known as transferred force [36]. This transferred force has different magnitude and direction at different parts of the compliant mechanism [37]. A specific member of the compliant mechanism behaves based on these properties of the transferred force acting on it [33]. Hence, the analysis of displacement of a specific member is enabled by transferred forces. Transferred force is mathematically derived and covered in further detail in [33]. The directional pattern of all the transferred forces from input to the output for the entire compliant mechanism is referred to as load flow [36].

3.2 LOAD TRANSMITTER-CONSTRAINT SET

The concept of transferred force enables the breakdown of a compliant mechanism into separate building blocks known as Load-Transmitter Constraint (LTC) Set [33]. This ensures an easy process of design analysis of a compliant mechanism. The design of a planar compliant mechanism mentioned in [38] is used as an example to illustrate the concept of LTC set. A Load Transmitter is a beam component of the compliant mechanism that allows load flow along its length [37]. A Constraint is another beam component of the compliant mechanism that only permits displacement along the perpendicular direction to its length [39, 40, 41, 42]. In Figure 3.1, it can be observed that the planar compliant mechanism comprises of four transmitters (T_1, T_2, T_3, T_4) and three constraints (C_1, C_2, C_3). The input applied force at point ***a*** causes an output displacement at point ***d.*** The direction of both input applied force (blue arrow in Figure 3.1) and output displacement (red arrow in Figure 3.1) are along positive Y-direction. The load flow direction along each beam component of the planar compliant mechanism is indicated by blue arrows in Figure 3.1. The reason behind this load flow direction is because of semicircular band generated by applied/transferred force, which is explained in Section 3.2.1.

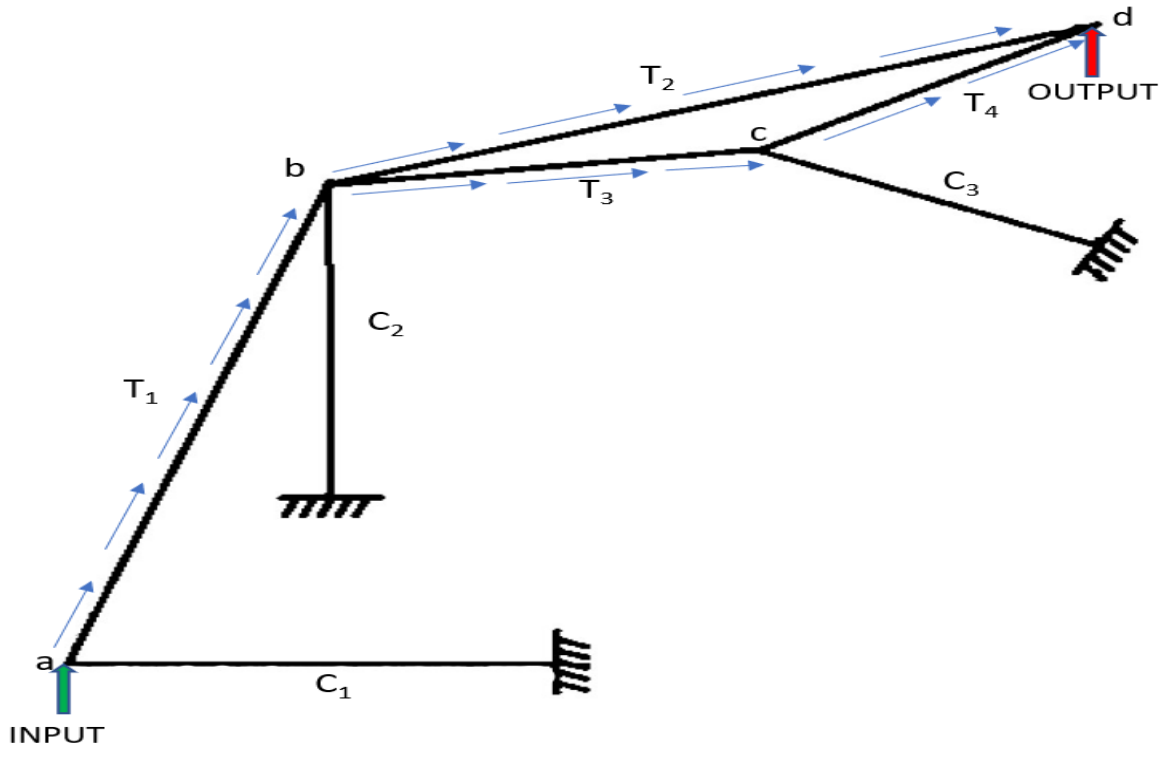


Figure 3.1: Planar Compliant Mechanism with load flow through its transmitters [38]

This compliant mechanism can be broken down into LTC sets as shown in Figure 3.2. The transferred force f_{trb} from point ***a*** to point ***b*** is dependent on Load Transmitter T_1 and Constraint C_1 only and not on transmitters T_2, T_3, T_4 and constraints C_2, C_3 (Figure 3.2(a)). This fact is mathematically evaluated in the Section 3.2.1. Similarly, the transferred load f_{trc} at point ***c*** is dependent on LTC Set 3 comprising of Load Transmitter T_3 and Constraint C_2 (Figure 3.2(c)). The input force for LTC Set 3 is f_{trb}, which means that the transferred force of the previous LTC set becomes the input force for the current LTC set in the sequence. Thus, the load flow in transmitter of a specific LTC set of a compliant mechanism is dependent only on applied force and transmitter orientation and independent of other LTC sets [36].

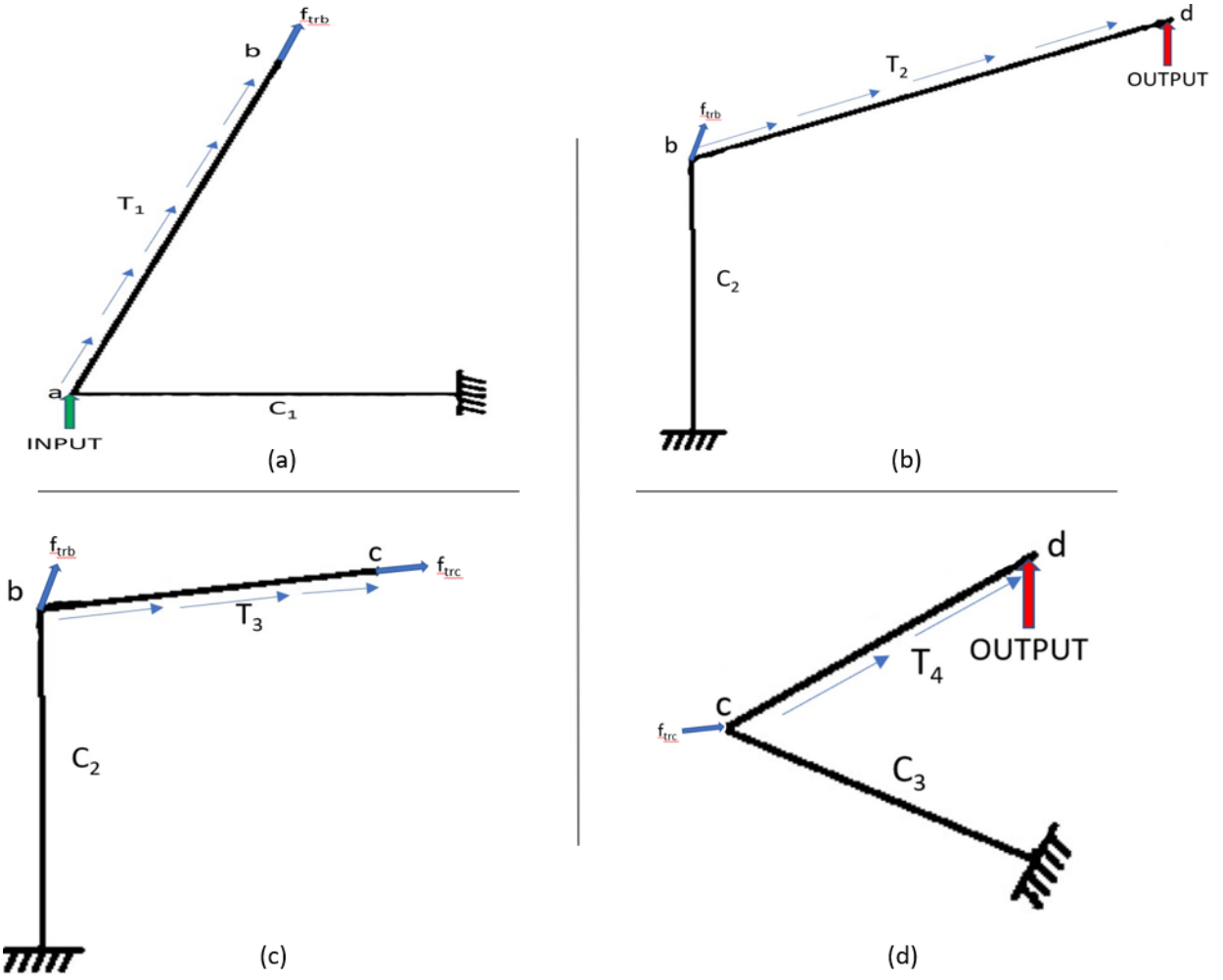


Figure 3.2: Planar Compliant Mechanism [38] broken down into (a) LTC Set 1 (b) LTC Set 2 (c) LTC Set 3 (d) LTC Set 4

3.2.1 PROPERTIES OF LTC SET:

A Compliant Dyad can be considered as single LTC set [36]. As an illustration, the first LTC set (LTC Set 1) in planar compliant mechanism, mentioned in Section 3.2, is considered as a Compliant Dyad (Figure 3.3). The length of constraint C_1 and transmitter T_1 are L_1 and L_2 respectively; while the second area moment are i_1 and i_2 respectively. The Young's Modulus for both C_1 and L_1 are represented by E.

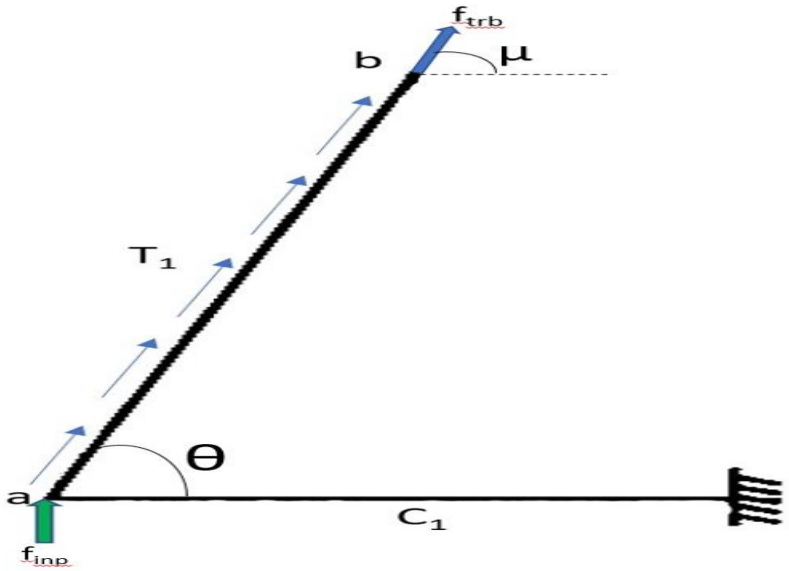


Figure 3.3: LTC Set 1 as a Compliant Dyad [38]

The following equation 1 of load transfer matrix L_T for a compliant dyad has been derived in [33]:

$$L_T = \begin{bmatrix} 0 & cot\theta & -\frac{[3(n_2+n_1^2\,cos\theta)]}{2(l_1 n_1(n_1+n_2)sin\theta)} \\ 0 & 1 & -\frac{[3(n_1)]}{2(l_1 n_1+l_1 n_2)} \\ 0 & 0 & -\frac{[(n_2)]}{2(n_1+n_2)} \end{bmatrix} \quad (1)$$

In the above equation, $n_1 = L_2/L_1$ and $n_2 = i_2/i_1$. The above load transfer matrix has been derived assuming that there is an input force and moment [33]. But in our case, there is only input applied force f_{inp} and the input moment is zero. Hence, all the elements in the third column of the above load transfer matrix are zero for the compliant dyad in Figure 3.3. Since there is no X component of f_{inp}, all the elements in the first column of Load Transfer matrix are zero. The direction of transferred force, due to the input applied force f_{inp}, is along the length of the transmitter T_1. The transferred moment at point ***b*** is zero since the element $L_{T3,2}$ in load transfer matrix is zero. The following equations 2,3 and 4 for transferred force have been derived in [33]:

$$f_{trbx} = cot\theta \times f_{inp} \quad (2)$$

$$f_{trby} = f_{inp} \quad (3)$$

$$\mu = \theta \quad (4)$$

In the above equations 2 and 3, f_{trbx} and f_{trby} are X and Y components of the transferred force f_{trb} at point ***b.***

In Figure 3.3, the output constraint C_2 at the point ***b*** was not considered because it does not have any impact on the load flow and its direction within the transmitter of LTC set [36]. The output constraint C_2 only has impact on the output displacement [36]. In Figure 3.4, there are two lines of displacement direction displayed at points ***a*** and ***b*** respectively. These displacement directions are perpendicular to the corresponding constraints. For instance, the displacement direction at point ***a*** is perpendicular to the constraint C_1. This direction of displacement is referred to as freedom line and it indicates the direction along which a point is free to move [39,40,41,42]. The displacement also depends on another factor called positive definiteness of stiffness matrix. This means that the angle between the direction of transferred force and output displacement at point ***b*** can have a maximum and minimum value of 90 and -90 degrees respectively [39,40,41,42]. This range of angles is denoted by semicircular band in Figure 3.4. Considering the above two

factors namely Constraint and positive definiteness of stiffness matrix, the final displacement direction of output at point ***b*** is along the coincidence of freedom line direction and semicircular band [36].

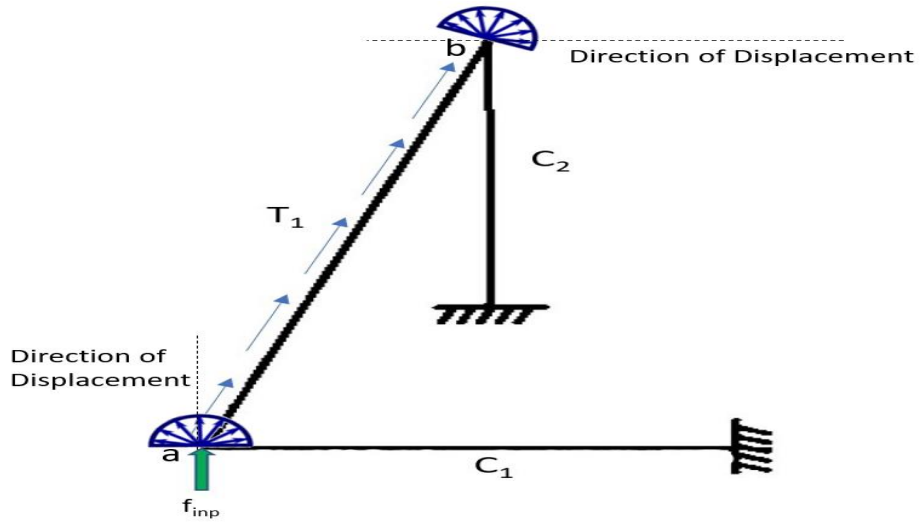


Figure 3.4: Semicircular bands and constraints of Compliant Dyad [38]

3.3 STEP BY STEP ALGORITHM FOR DESIGN SYNTHESIS

The design guidelines for planar compliant mechanisms and mechanical metamaterials, validated by Krishnan et al. and Patiballa et al. respectively [37, 34, 35], is applied and demonstrated on a simple planar compliant mechanism in this section. The considered design for explaining the algorithm is a Single Input-Single Output Planar Compliant mechanism. The design domain, on which this compliant mechanism is designed, is a square with some boundary conditions. The given specifications include input and output points being placed on the opposite sides of the design domain, wherein both these points are placed on the end points of the left vertical side of the domain. The displacement of the input point is along the positive Y direction; while the required displacement of the output point is along the negative Y direction. So, the input displacement direction is inverted at the output point. As mentioned in earlier sections, the transferred force moves along the length of transmitter only if there is applied/transferred force. Hence, the applied/transferred moment are not acting on any of the components in this design.

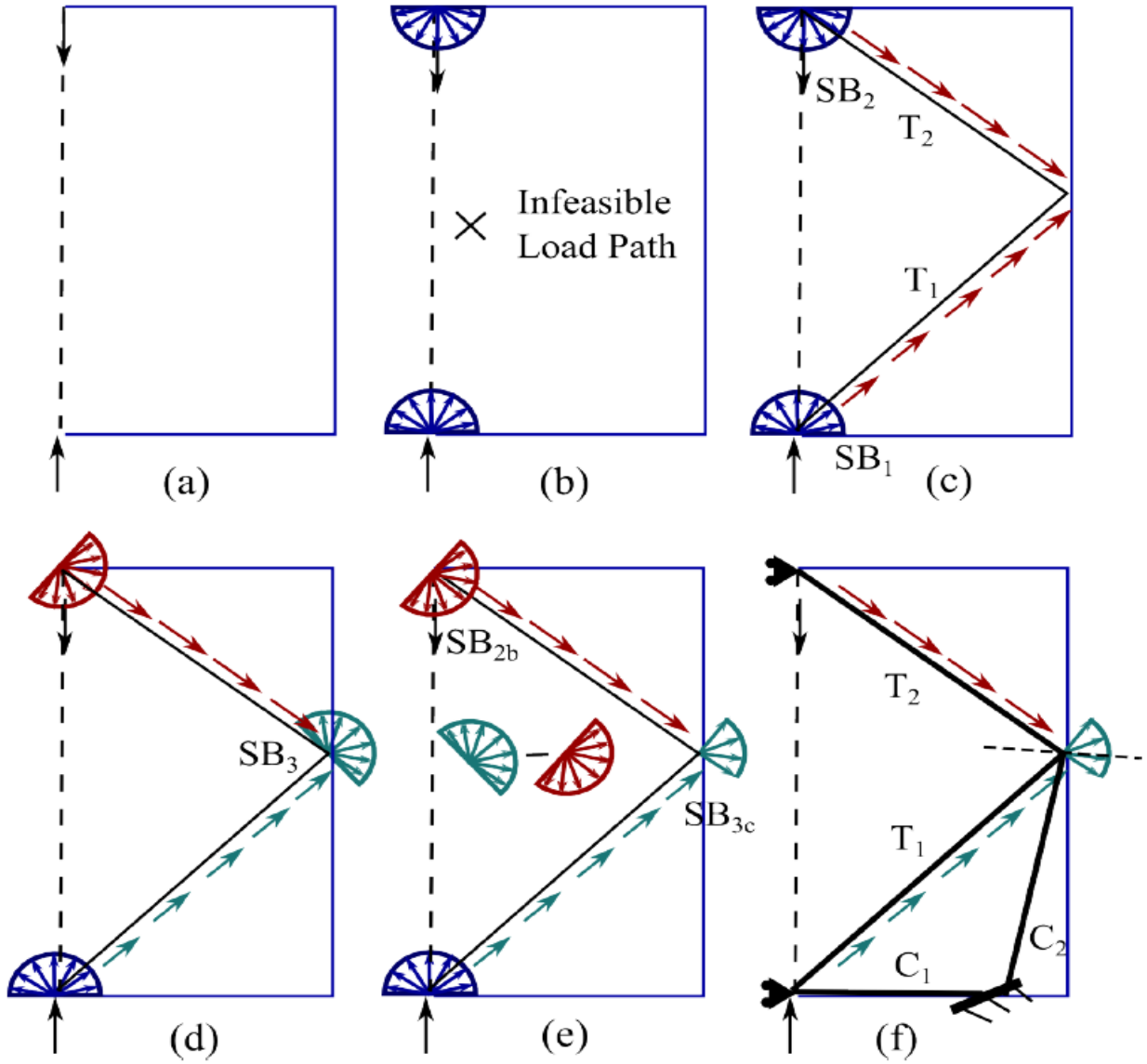


Figure 3.5: Design steps [37] for single input single output compliant mechanism. (a) Direction of displacement at input and output points respectively (b) Infeasible load path when there is a single transmitter connecting input and output points, considering the orientation of semicircular bands (c) Transmitters T_1 and T_2 are aligned such that the load flow within them are one of the possible force directions within SB_1 and SB_2 respectively (d) The modified semicircular band SB_3 due to the direction of load flow in transmitter T_1 (e) The intersected semicircular band SB_{3c} at the intermediate point, resulting from the intersection of modified semicircular bands SB_{2b} and SB_{3c} (f) The orientation of constraints C_1 and C_2 at input and intermediate points respectively.

STEP 1: ORIENTATION OF SEMICIRCULAR BANDS AT INPUT AND OUTPUT POINTS

The possible range of direction for input and output forces is from -90 degrees to +90 degrees with respect to the direction of input and output displacements respectively. This range of force direction that perform positive work is depicted in the form of a semicircular band in Figure 3.5 (b).

The load flow direction within the transmitter has to be one of the possible directions in both SB1 and SB2 at input and output points respectively [34, 35, 37]. So, the load path consisting of single transmitter, depicted in Figure 3.5 (b), is not possible because the positive Y-direction of load flow from SB1 is not part of any of the directions in SB2. Thus, there is a need for at least two transmitters for the given design.

STEP 2: LOAD FLOW DIRECTION WITHIN TRANSMITTER

Though there can be any number of transmitters within a design domain, the considered design has two transmitters T_1 and T_2 between input and output point to keep the design synthesis simpler. These two transmitters can be inclined in any direction within the design domain as long as these two transmitters have equal length [37]. In Figure 3.5 (c), the load flow direction follows one of the possible directions within the semicircular bands SB1 and SB2. If the number of transmitters is greater than two, then the transmitters passing through SB1 and SB2 need to have a load flow direction along one of the respective directions within these semicircular bands. While the other transmitters between them can have any load flow direction as long they remain in tension and do not buckle [37].

STEP 3: INTERMEDIATE POINT AND ITS TRUNCATED SEMICIRCULAR BAND

Based on the load flow direction derived in the previous step, the modified semicircular bands are extracted [34, 35]. In Figure 3.5, the modified semicircular band SB_3 has its base line oriented perpendicular to the load flow direction in transmitter T_1. Similarly, the modified

semicircular band SB_{2b} has its base line oriented perpendicular to the load flow direction in transmitter T_2. The intersection point between the two transmitters T_1 and T_2 is referred to as intermediate point. A truncated semicircular band is generated at the intermediate point. The truncated semicircular band is the intersection of the two modified semicircular bands [37] namely SB_3 and SB_{2b} as shown in Figure 3.5. The possible range of displacement directions of intermediate point is depicted by the truncated semicircular band [34, 35].

Step 4: ORIENTATION OF CONSTRAINT AT THE INTERMEDIATE POINT

The designer can pick any of the possible displacement directions within the truncated semicircular band. The line perpendicular to the constraint line C_2 is referred to as freedom line and it needs to be oriented along one of the directions within the truncated semicircular band (Figure 3.5) [37, 41, 42]. The constraint C_1 makes sure that the input does not move in X direction. Due to the symmetry of the compliant mechanism, the output is constrained in X direction too.

CHAPTER 4: DESIGNING SCM USING VR SOFTWARE

This chapter initially covers the Step-by-Step algorithm for designing a simple spatial compliant mechanism using VR. It is followed by validation of the algorithm on other similar spatial compliant mechanisms.

4.1 STEP-BY-STEP ALGORITHM FOR DESIGN OF SPATIAL COMPLIANT MECHANISMS

The step-by-step algorithm for spatial compliant mechanisms is an extension of the design guidelines for planar compliant mechanisms and mechanical metamaterials validated by Krishnan et al. and Patiballa et al. respectively [37, 34, 35]. There are minor changes made to the guidelines for planar compliant mechanism, so that they can be accommodated for spatial compliant mechanisms. The considered design for explaining the algorithm is a Single Input-Single Output Spatial Compliant mechanism. The design domain, on which this compliant mechanism is designed, is the transparent cube covered in Chapter 2. The given specifications include input and output points being placed on the diagonally opposite edges of the design domain, wherein both these edges are the intersection line between XY and YZ planes. The force applied on the input point is along the positive X direction; while the required displacement of the output point is along the positive Z direction. The intermediate point, intersection point between two transmitters, lies near the center of the design domain.

STEP 1: PLACEMENT OF INPUT, INTERMEDIATE AND OUTPUT POINTS

As the user place their left thumb on the thumb stick of the left joystick of Oculus Touch Controllers [46], the Compliant tools menu pops up. In Figure 4.1, the torus shaped menu pops up over the virtual Oculus Touch Controllers [46] and it is the Compliant tools menu.

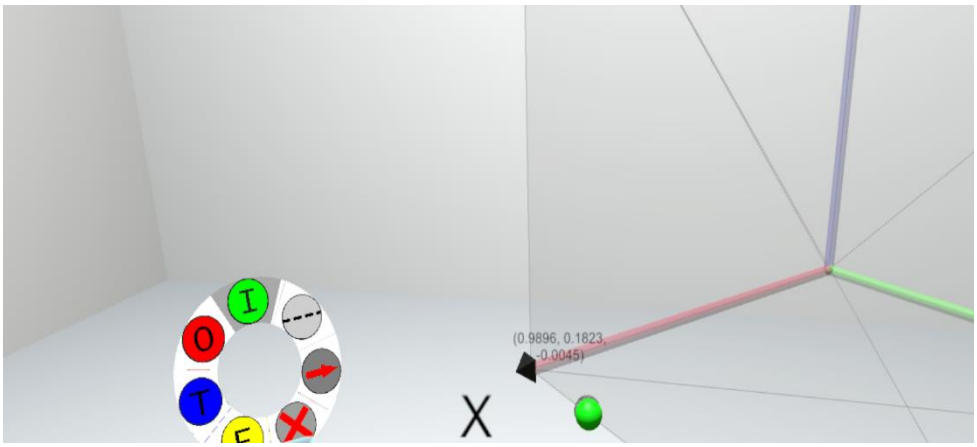


Figure 4.1: Placing Input point sphere (green) on the design domain by pressing 'I' button on Compliant tools menu

The user can navigate to the 'I' button on the tools menu using the left joystick and press it to select the Input tool. Then an Input point (green colored sphere) appears on a specific face of the transparent cube (design domain). The input point follows the user's movement of Oculus Touch Right Controller [46] on the design domain. The input point can be placed at a desired location on the domain by pressing button A on the Oculus Touch Right Controller [46].

Similarly, most of the above functions apply to Output (denoted by 'O') and Intermediate (denoted by 'T') points (Figure 4.2). The only difference is that there is no point proximity function for an intermediate point because it needs to be placed inside the volume of the cube and its location can be anywhere within the volume.

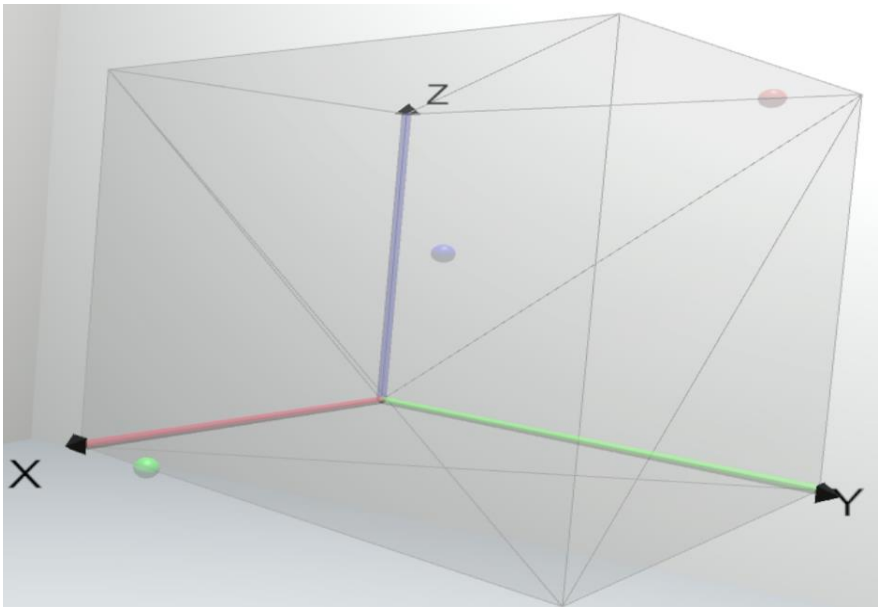


Figure 4.2: Placement of Input (Green sphere), Intermediate (Blue Sphere) and Output (Red Sphere) points

STEP 2: INPUT FORCE AND OUTPUT DISPLACEMENT

This function can be activated by pressing the button with a red arrow within the tools menu. Once an input/output point is placed on the cube, the force/displacement button can be pressed and a 'force/displacement sphere' appears. The force/displacement sphere is used to draw trace of force/displacement vector from its tail to its head arrow. The starting point of the force/displacement should be placed on the input/output point respectively and the ending point of the force/displacement sphere can be placed anywhere around the input/output point respectively (Figure 4.3). This leads to an outward force/displacement from the input/output point respectively. If a force/displacement vector needs to be drawn inward towards the input/output point respectively, the reverse operation of the previous action needs to be performed. This might be harder if the user wants to draw an inward force/displacement vector exactly on X, Y or Z coordinate axes. Hence, an 'axis locking' mechanism is introduced, and it can be activated by pressing the right joystick after the force function is activated. For instance, if the user wants to draw an inward force vector for input point in X direction, then the user can start anywhere close to the 90 degrees with respect to Y/Z direction and end over the input point. The software algorithm

behind the axis locking mechanism corrects the force vector so that it is exactly 90 degrees with Y/Z direction and makes sure that there are no human errors in drawing the force vector. This is an essential feature because it is hard to draw a vector with precise angle by free hand. The input force vector is denoted by blue color, while the output displacement vector is denoted by red color.

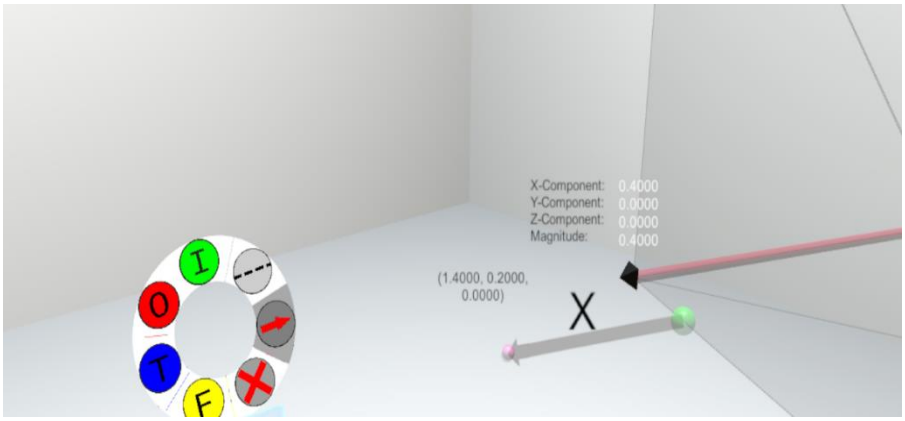


Figure 4.3: Drawing Force vector on input point by dragging right controller [46] away from input point after pressing Arrow (Red) button on Compliant Tools menu

STEP 3: CONSTRAINTS AT INPUT AND OUTPUT POINTS

Based on the direction of force and displacement vectors at input and output points respectively, two constraints are applied each on the input and output points along the perpendicular axes to that of their force/displacement vector. In this example, the force vector on the input point is applied along the positive X direction. Hence, the two constraints are applied along Y and Z axes respectively (Figure 4.4). These constraints ensure that both input and output points move only along the direction of the force and displacement vectors respectively and not along its perpendicular axes.

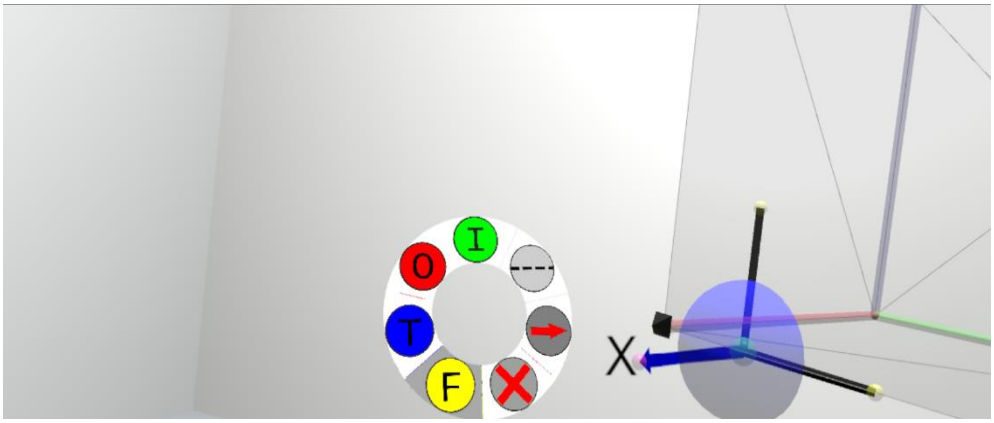


Figure 4.4: Constraint lines are denoted by black lines between input (green) and fixed (yellow) point spheres. It can be drawn using 'F' button on Compliant Tools menu

STEP 4: HEMISPHERICAL BANDS OF INPUT AND OUTPUT POINTS

The hemispherical band is generated, by the software, immediately after the force/displacement vector is drawn for the respective input/output point. The hemispherical band represents the possible directions of input displacement and it can range from ± 90 degrees with respect to the applied force direction at the input [37]. Similarly, the hemispherical band at the output point represents the possible directions of output transferred force and it can range from ± 90 degrees with respect to the required output displacement [37]. The base circle of the hemisphere band is placed perpendicular to the force/displacement vector. As it can be observed in the Figure 4.5, the hemispherical band for the input and output points are facing in Y and Z directions respectively.

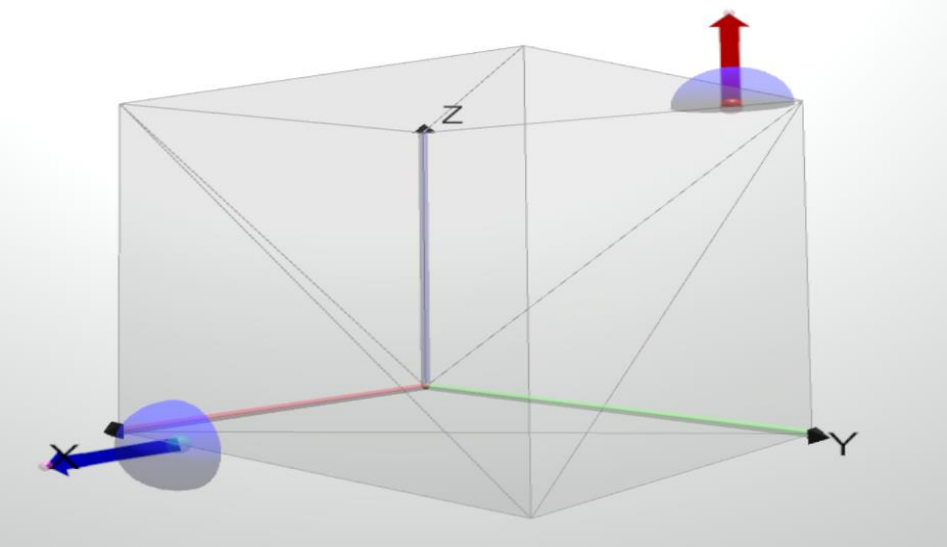


Figure 4.5: Hemispherical bands at input and output points

STEP 5: TRANSMITTERS AND TRUNCATED BAND

A single transmitter cannot be used to make a connection between the input and output point because the hemispherical band of the input point creates a load flow in the transmitter that has a direction towards the input point. This load flow direction does not correspond to any of the possible directions in the hemispherical band of the output point. Hence, there is a need for at least two transmitters for this design. The intersecting point between these two transmitters is referred to as intermediate point. Transmitter is drawn between input and intermediate points by selecting button 'T' on the Compliant Tools menu. Then the user needs to press the button A on right controller [46] and drag the right controller [46] from input point sphere to the intermediate point sphere. On reaching the intermediate point, the button A needs to be released and a black line (transmitter) appears between input and intermediate points (Figure 4.6). Similarly, another transmitter is drawn between intermediate and output point.

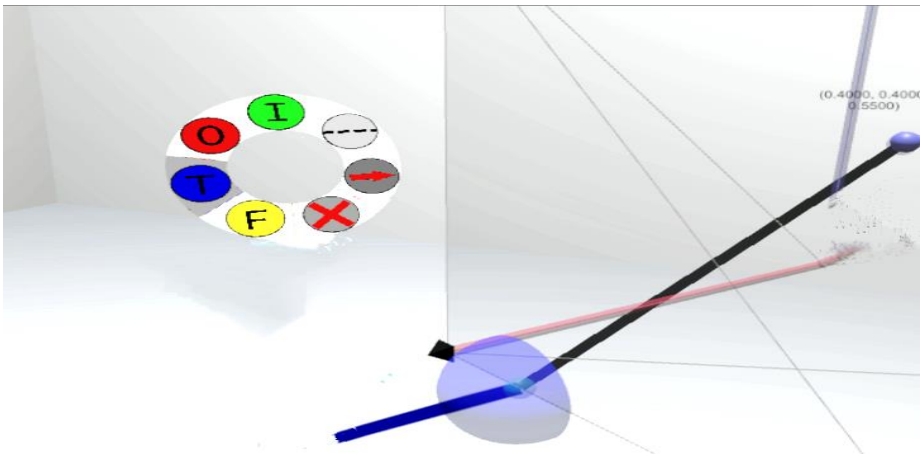


Figure 4.6: Drawing Transmitter from Input to Intermediate Point using right controller [46] after pressing ‘T’ button on Compliant Tools menu.

The transmitter between the input and intermediate points has a load flow direction towards the input point (Figure 4.7). While the transmitter between the output and intermediate point has load flow direction towards the output point (Figure 4.7). The load flow direction is chosen such that there is no unwanted bending of the transmitters [37]. The modified hemispherical bands at the input and output points are established based on the transmitter orientation [37] (Figure 4.8). The truncated band at the intermediate point is, an intersection of modified hemispherical bands at input and output points, depicting the possible directions of displacement of the intermediate point [34, 35]. The truncated band will be automatically generated by the software after the creation of intermediate point (Figure 4.9).

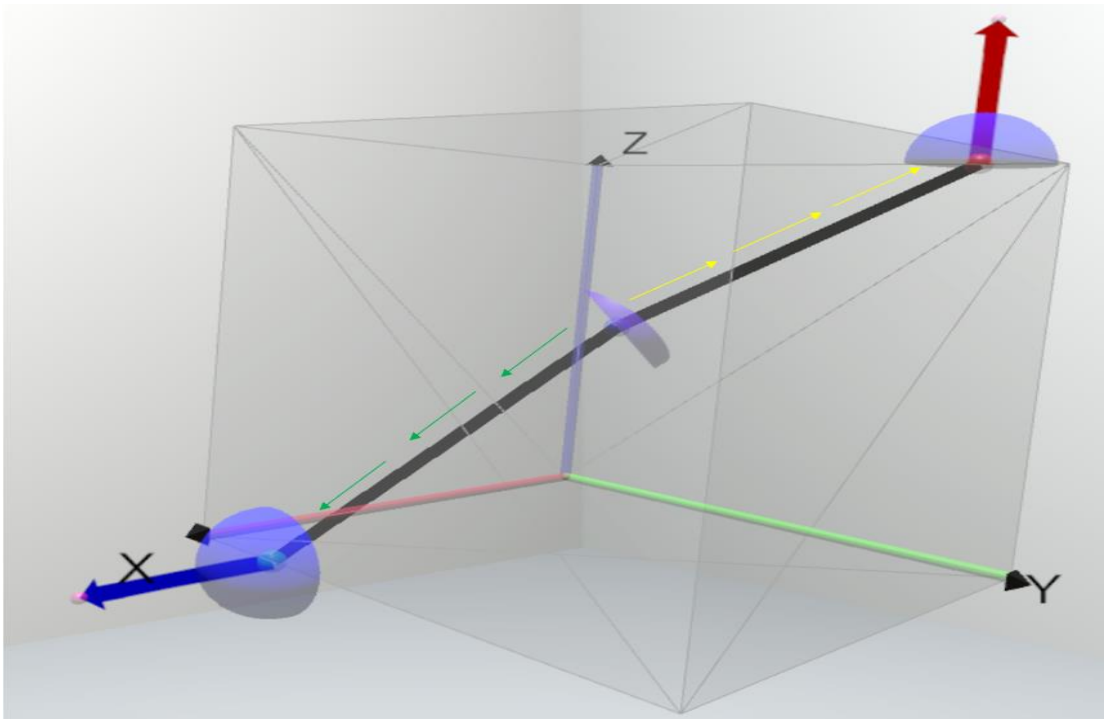


Figure 4.7: Load flow directions, towards input and output points, on transmitters are represented by green and yellow arrows respectively

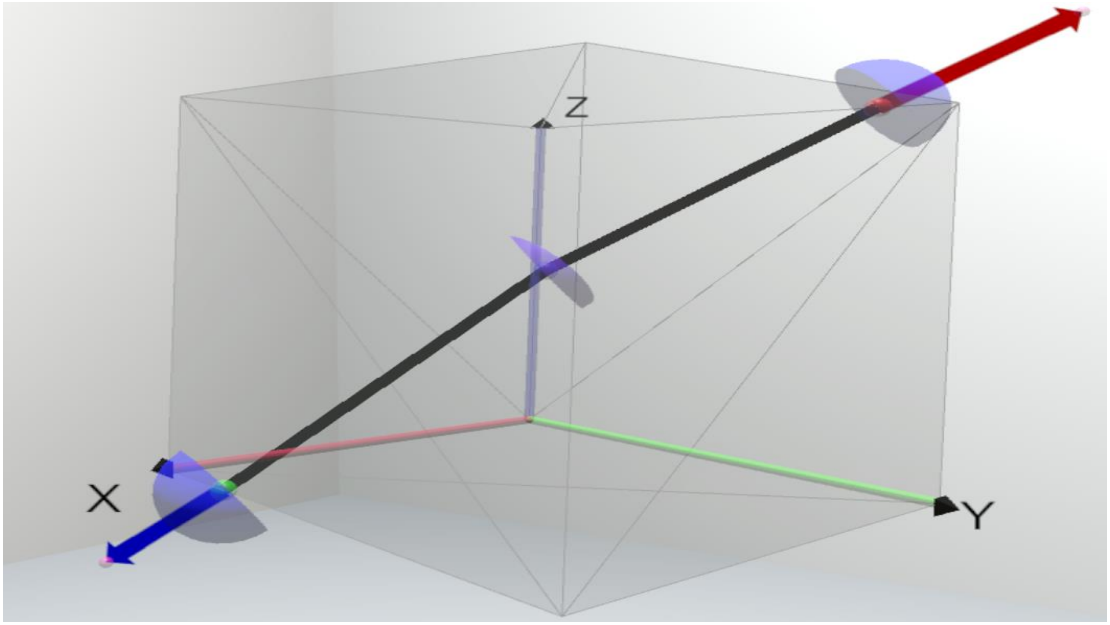


Figure 4.8: Modified hemispherical bands with displacement vector (blue arrow) and transferred force vector (red arrow) at input and output points respectively

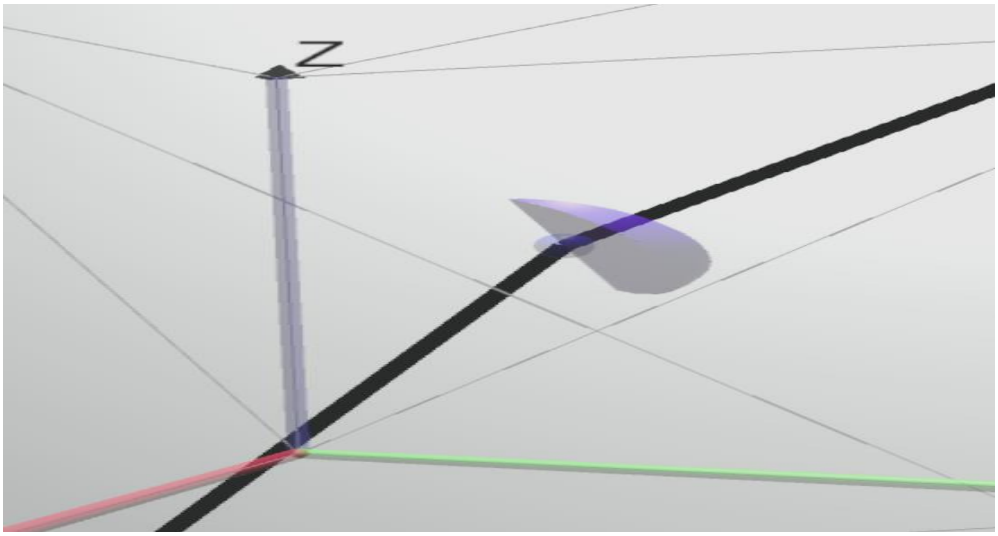


Figure 4.9: Truncated band at intermediate point, resulting from intersection of modified hemispherical bands of input and output points

STEP 6: FREEDOM LINE

A Freedom line is a 3D ray, drawn within the region of truncated band, for finalizing the displacement direction of the intermediate point. This is the beginning of an important phase of design synthesis through Freedom and Constraint topologies for Compliant mechanisms [41, 42]. The freedom line is drawn from the base of the truncated cone towards the curved portion of the cone. It is denoted by the dotted lines (Figure 4.10) and the user needs to select ‘dotted line’ tool on the Compliant Tools menu. Then press the button A on the right controller [46] at the starting point of the freedom line; hold the button and stretch along the desired direction and release the button. In this example, the freedom line is drawn along the positive Y direction.

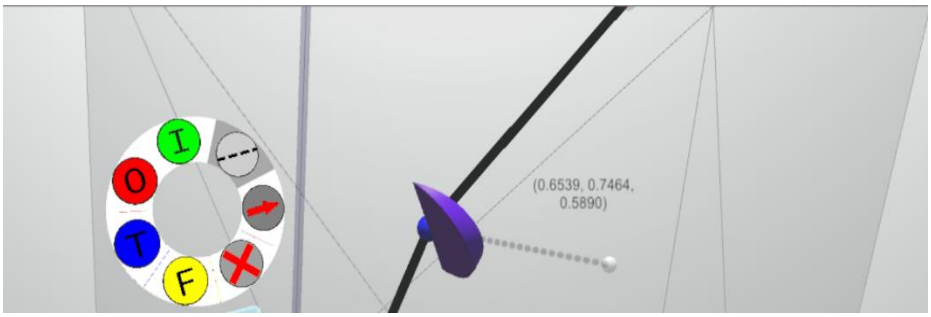


Figure 4.10: Drawing Freedom Line from the truncated hemisphere attached to intermediate point by dragging right controller [46] after pressing dotted line button on Compliant tools menu

STEP 7: CONSTRAINT PLANE

The constraint plane decides the displacement direction of the intermediate point, based on the load flow direction in both the transmitters [34, 35]. A constraint plane is automatically generated by the software, once the freedom ray is chosen. Then, the user can draw two constraint lines on this plane, which are both perpendicular to the freedom ray. The algorithm prevents the user from drawing a constraint line, that is going outside the domain space, by representing it as a red line. Any such red line will not be placed on the plane eventually by default. The accepted constraint line is represented by a white line on the constraint plane. In this example, the constraint lines are drawn in diagonally opposite directions from the center of the plane (Figure 4.11).

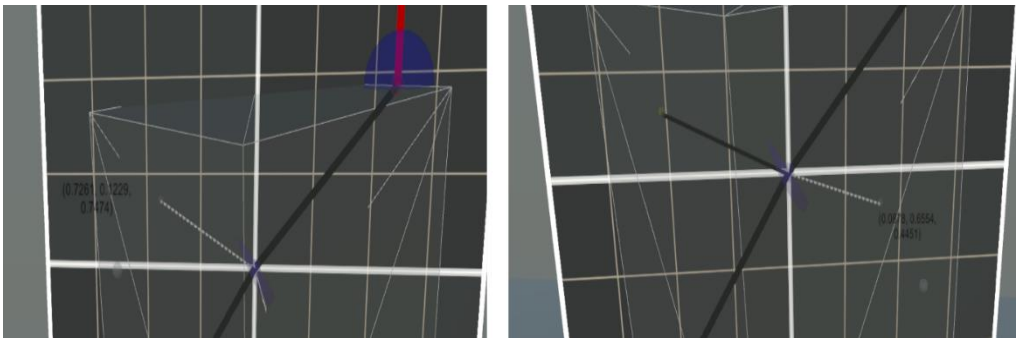


Figure 4.11: Two constraint lines (dotted lines) drawn on the constraint plane (checked square) by dragging right controller [46] away from intermediate point

STEP 8: DESIGN SUBMISSION TO MATLAB

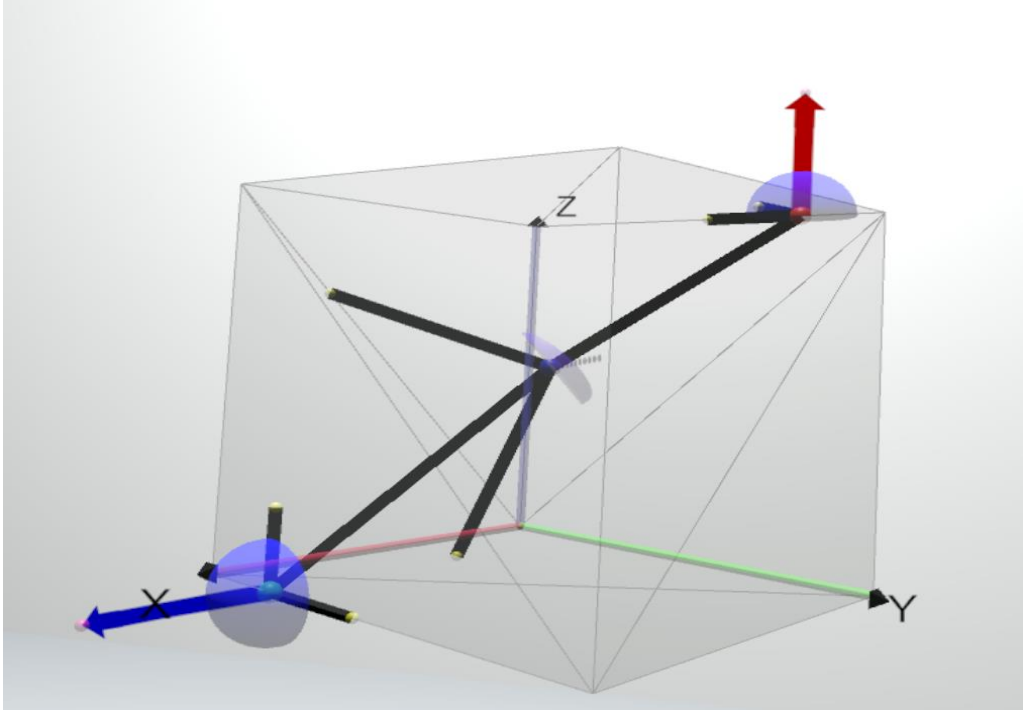


Figure 4.12: The finalized compliant mechanism design ready for design analysis

After the constraint lines are drawn, the compliant design is ready for submitting to Matlab [51] for design analysis (Figure 4.12). The design data can be submitted to Matlab [51] by pressing button B on the right controller [46]. Then, a green colored ray cast is released from the right virtual hand on to the "Submit Design" button on one of the walls of the room. This button will be highlighted in yellow color, once the ray cast is concentrated on it and then the fire button on right controller [46] needs to be pressed, in order to press the Submit Design button (Figure 4.13). Then the Design data will be transferred from the Unity [43] software to Matlab [51].

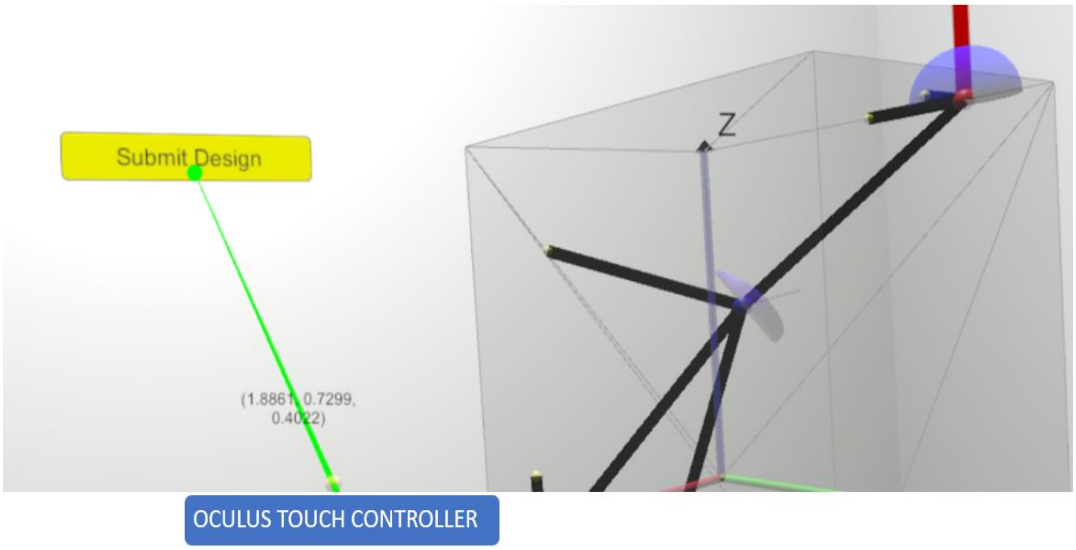


Figure 4.13: Shooting green raycast from Oculus Touch Controller [46] to submit design to Matlab [51]

STEP 9: DESIGN ANALYSIS IN MATLAB

The Compliant Mechanism analysis algorithm was an in-house three-dimensional Finite Element code written in Matlab [51] by Patiballa. This algorithm is applied on the design made in the VR Unity [43] software. As it can be observed in Figure 4.14, the Matlab [51] generates a plot of deflected compliant mechanism design after various forces are applied on different points. This deflection data is transferred back to Unity [43] automatically due to the Unity [43]-Matlab [51] Network Integration algorithm.

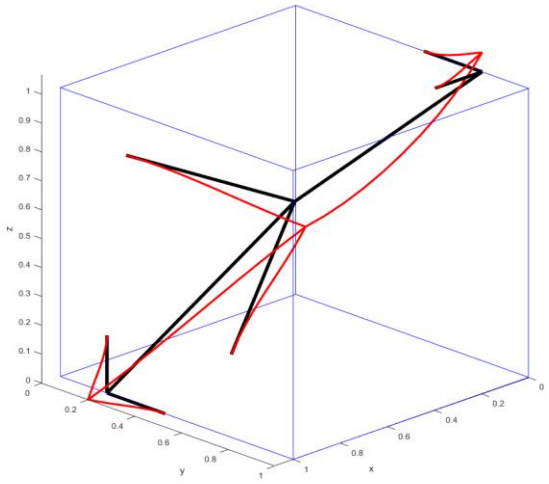


Figure 4.14: Matlab [51] plot after deflection of the compliant design, wherein the red lines denote the deflected compliant design and the black lines denote the original compliant design

STEP 10: MODIFIED COMPLIANT DESIGN IN UNITY

After the modified data of the compliant design is sent back to Unity [43], the deflected design of the compliant mechanism is displayed along with the original design. This three-dimensional representation gives the scope for a qualitative analysis of deflection across various points of the compliant design. In Figure 4.15, it can be observed that the original compliant design is represented by black lines, while the deflected design is represented by red lines.

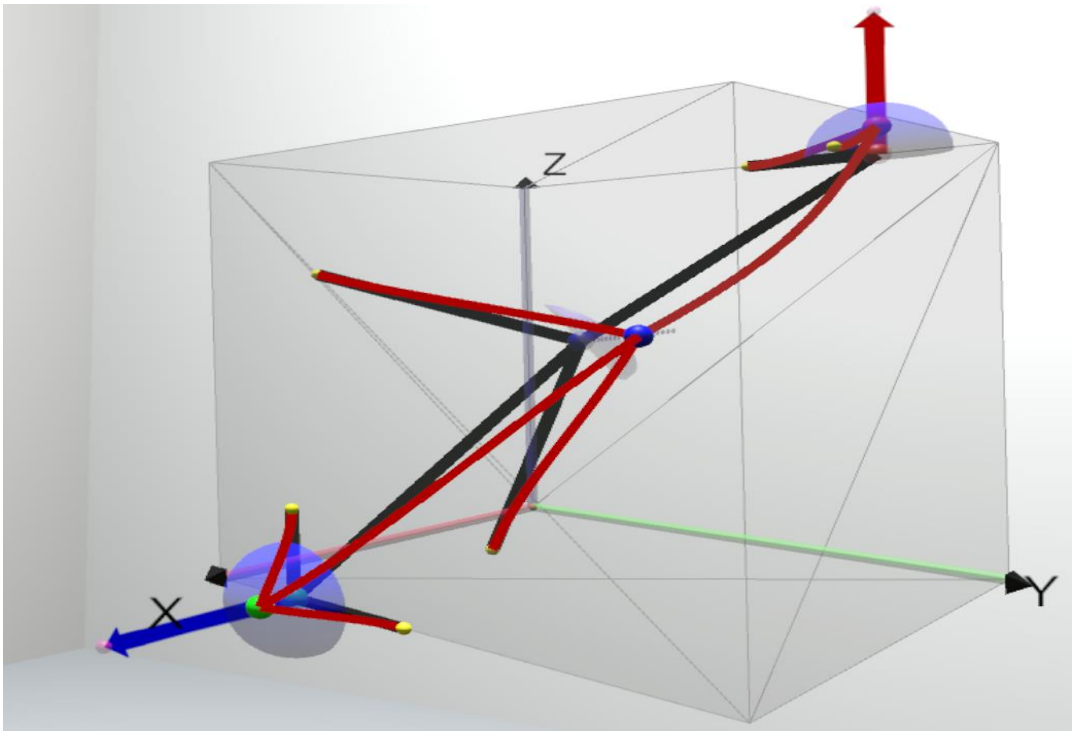


Figure 4.15: Final Design Model in VR after analysis in Matlab [51], where original compliant design is represented by black lines, while the deflected design is represented by red lines

4.2 VALIDATION OF SOFTWARE USING SINGLE INPUT-SINGLE OUTPUT SPATIAL COMPLIANT MECHANISMS

In order to verify if the step-by step design guidelines worked on the VR software application, five simple spatial compliant mechanisms were considered. These spatial compliant mechanisms are single input-single output designs, similar to the example illustrated in Section 4.1. Figures 4.16-4.25 show the final output of compliant mechanism before and after Finite Element analysis for all the five designs in a sequence. The transmitters and constraints are depicted by black and red lines for compliant mechanism before and after Finite Element analysis. The common elements of these five designs include:

a) Input position at (1,0,0) [considering the unit length of a side in cube]
b) Direction of applied force at input point i.e. positive Z direction
c) Input constraints in X and Y directions respectively
d) Intermediate point almost at the center of design domain (transparent cube)

e) Intermediate constraints at X and Z directions respectively

f) Output position at (0,1,1).

4.2.1 DESIGN 1:

Output Displacement: Negative Z Direction

Output Constraints: X and Y direction

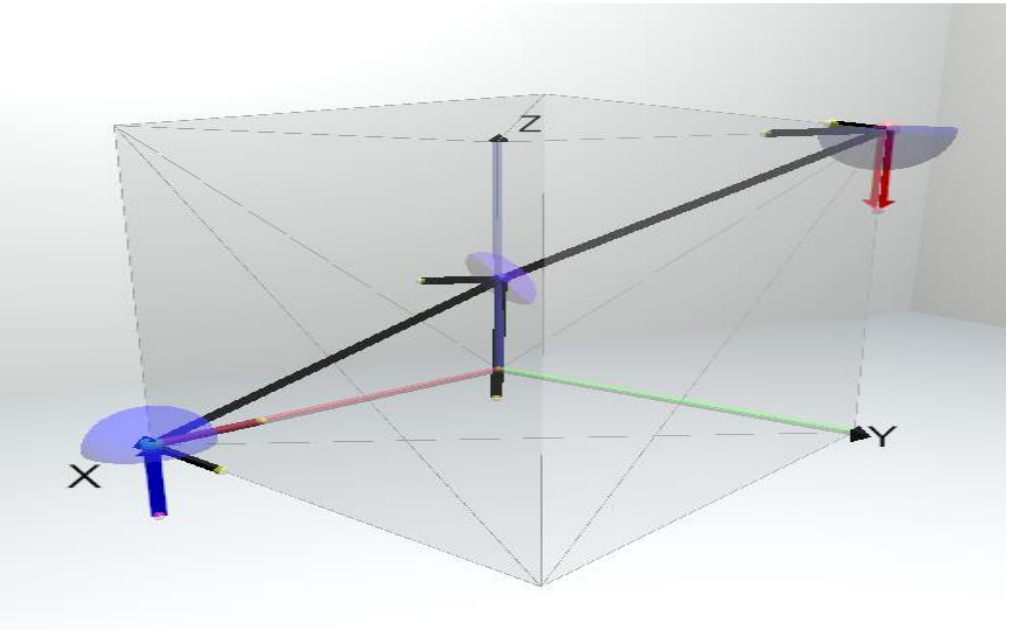


Figure 4.16: Spatial Compliant Mechanism Design 1 before 3D Finite Element analysis

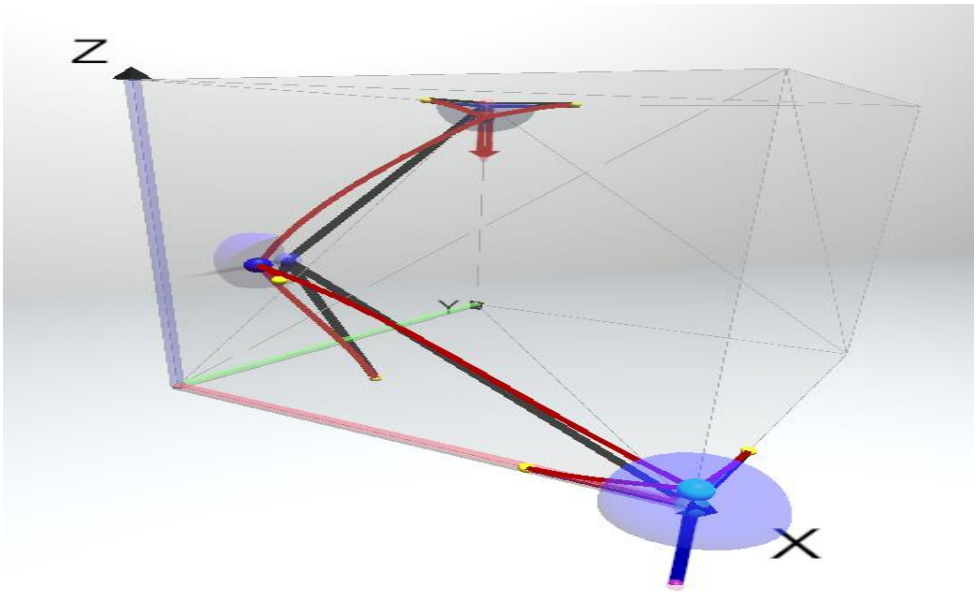


Figure 4.17: Spatial Compliant Mechanism Design 1 after 3D Finite Element analysis

4.2.2 DESIGN 2:

Output Displacement: Positive X Direction

Output Constraints: Y and Z direction

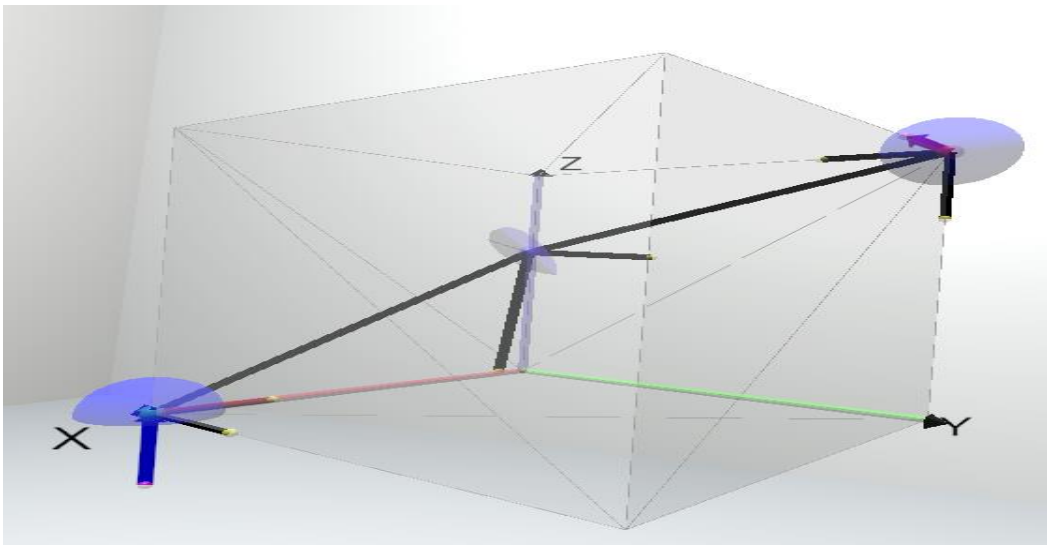


Figure 4.18: Spatial Compliant Mechanism Design 2 before 3D Finite Element analysis

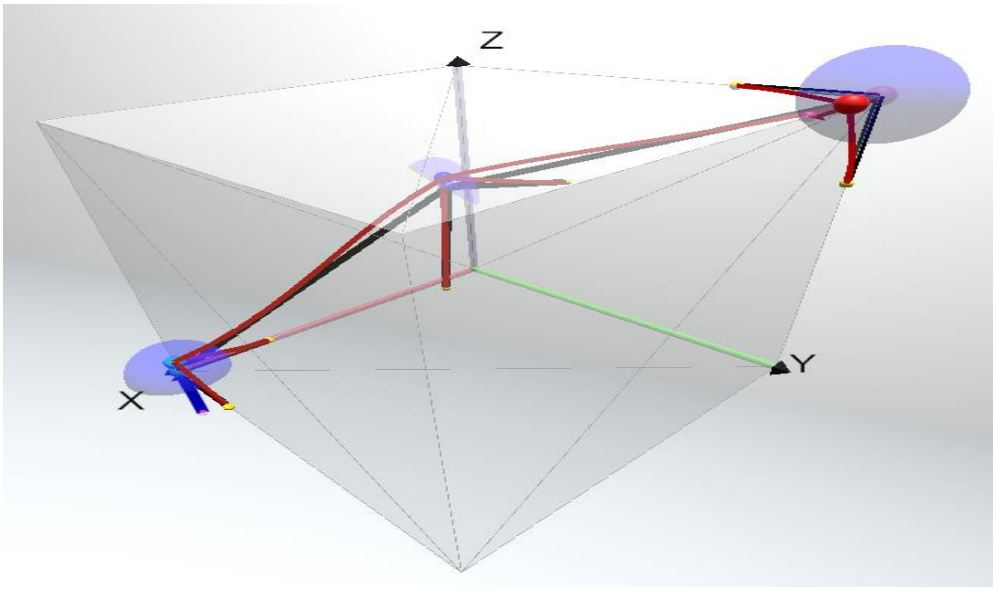


Figure 4.19: Spatial Compliant Mechanism Design 2 after 3D Finite Element analysis

4.2.3 DESIGN 3:

Output Displacement: Negative Y Direction

Output Constraints: X and Z direction

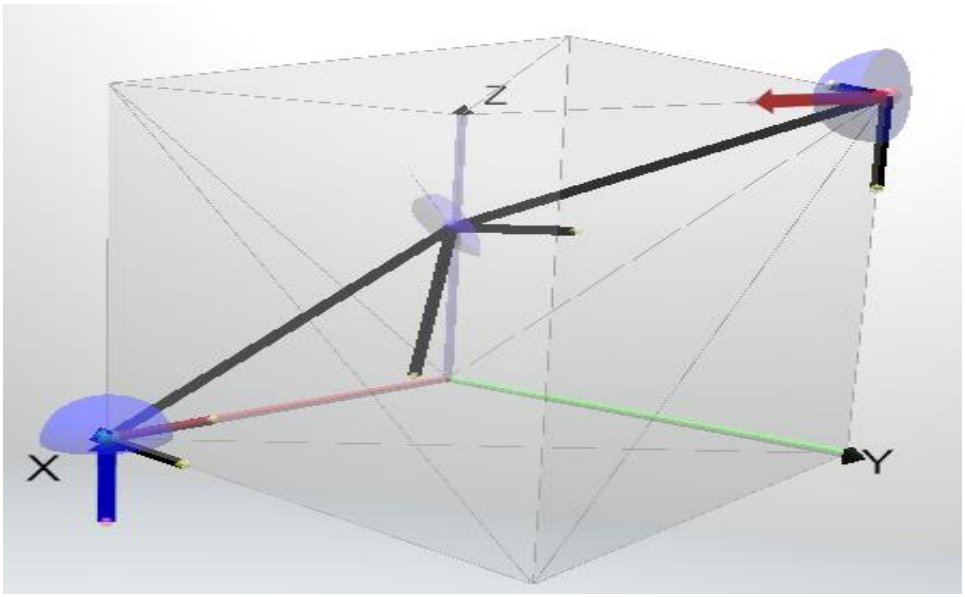


Figure 4.20: Spatial Compliant Mechanism Design 3 before 3D Finite Element analysis

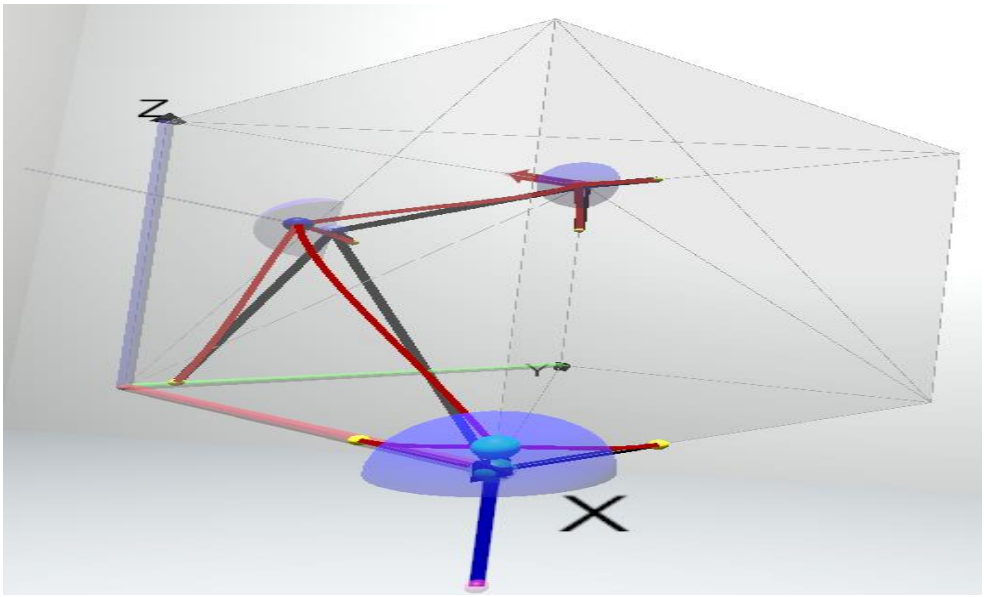


Figure 4.21: Spatial Compliant Mechanism Design 3 after 3D Finite Element analysis

4.2.4 DESIGN 4:

Output Displacement: Negative X Direction

Output Constraints: Y and Z direction

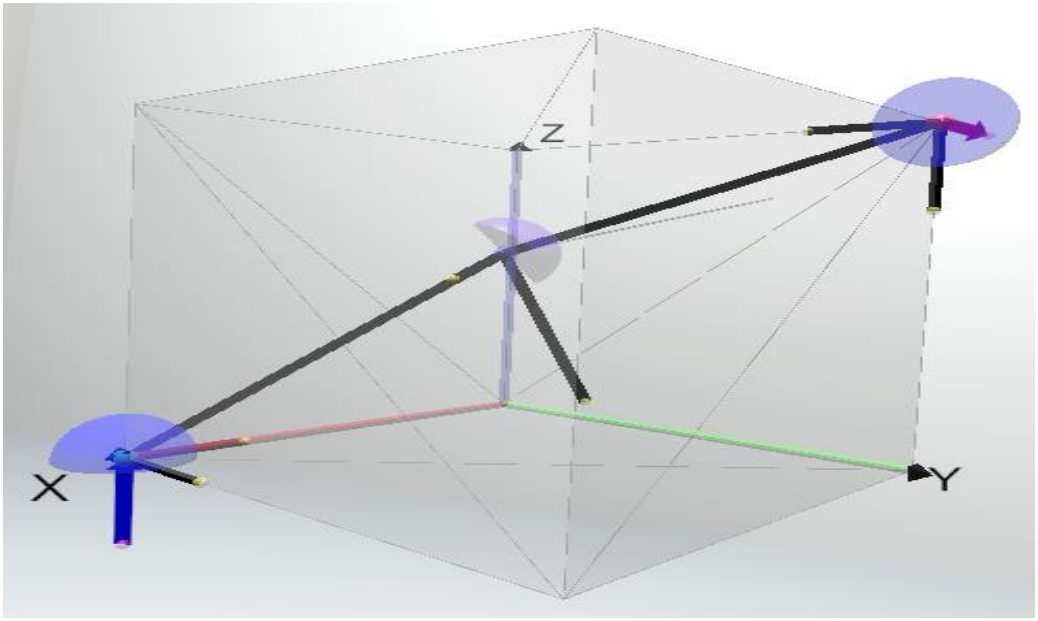


Figure 4.22: Spatial Compliant Mechanism Design 4 before 3D Finite Element analysis

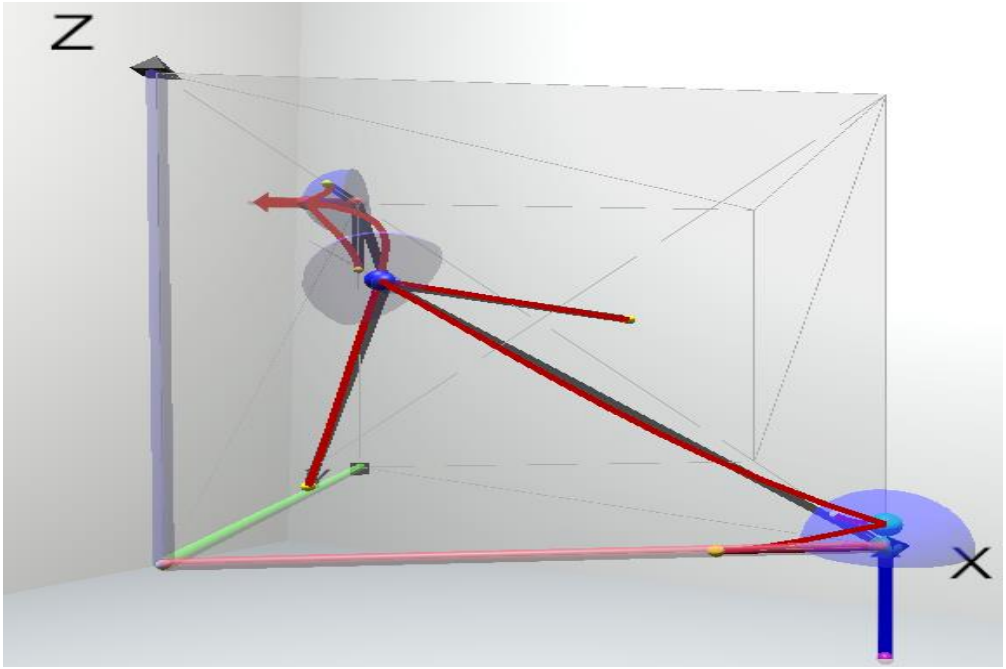


Figure 4.23: Spatial Compliant Mechanism Design 4 after 3D Finite Element analysis

4.2.5 DESIGN 5:

Output Displacement: Positive X Direction

Output Constraints: Y and Z direction

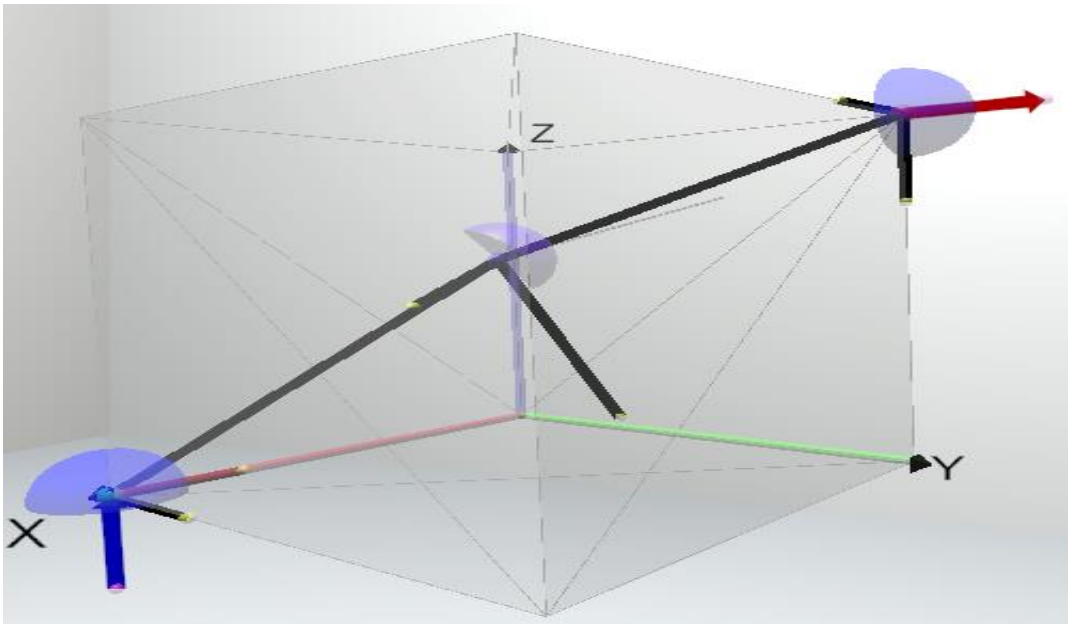


Figure 4.24: Spatial Compliant Mechanism Design 5 before 3D Finite Element analysis

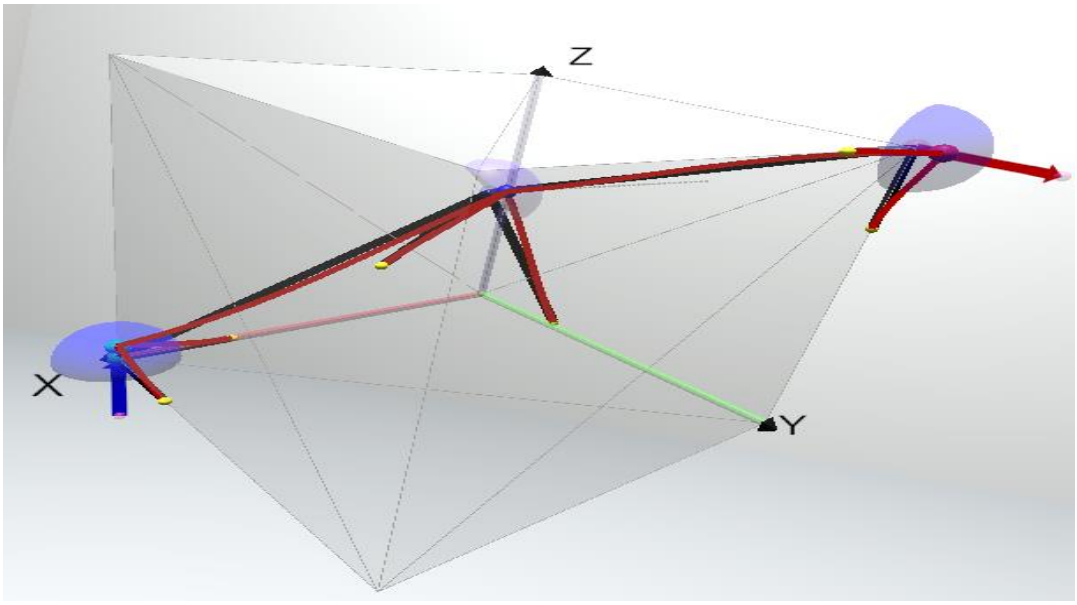


Figure 4.25: Spatial Compliant Mechanism Design 5 after 3D Finite Element analysis

4.2.6 KEY INFERENCES FROM FIVE DESIGNS:

1. All the five spatial compliant mechanisms were successfully validated on the VR software, using the three-dimensional design guidelines.

2. For the same design problem of a SCM, there can be multiple alternate solutions by changing the number of transmitters and/or intermediate points.

3. The deflection direction of intermediate point depends on the orientation of the constraint plane with respect to the point.

CHAPTER 5: CONCLUSIONS AND RECOMMENDATIONS

5.1 CONCLUSIONS

The objective of this study was to develop a Virtual Reality (VR) design software platform that enables proper three-dimensional visualization of some of the design guidelines for SCM, that were challenging to implement load flow visualization method on pen and paper. Some of the challenges include the three-dimensional visualization of the input/output hemispherical bands; truncated hemispherical band (intersection between the modified input and output hemispherical bands); freedom ray and constraint plane. A three-dimensional VR design platform had been established for designing SCM using load flow visualization method in this thesis research.

The architectural framework necessary for the creation and execution of various key features of the design software was developed and validated. Based on the software development tools available in the version 5.6.2f1 of Unity 3D [43] game engine, this architectural framework was conceptualized. In addition, the Unity 3D [43] assets that were used in the software development include VRTK [44] (Virtual Reality Toolkit); Vectrosity [45]. The design software is user-friendly because it is easily customizable, according to the needs of the user. The user can make easy and quick changes to the software so that it accommodates the application of a desired design method for SCM. The essential features that enhanced the three-dimensional visualization of the design modeling as well as provided an immersive experience to the user include:

a) The fixation of design domain in three-dimensional space, so that the user can walk around the design and get its different view perspectives. This is crucial for design analysis.

b) The design domain can also be scaled up/down in all directions and its orientation can be changed in any direction. This is an important feature because scaling enables the user to analyze the deformation of a compliant mechanism design in a deeper perspective.

c) The text showing the coordinates of a point, above the right Touch Controller [46], gets updated in real time, as the user moves the point around different parts of design domain, using the right Touch Controller [46]. This is a useful feature because it helps the user to place a desired point in the exact location, if the user knows the coordinates prior to the design modelling.

d) The point proximity feature ensures that the user doesn't have to grab the design domain and place the point in the desired location. The user can instead just move the right touch controller [46] and the point proximity algorithm tracks the user's hand movements and shows the current location of the point on the closest face of the domain. Point proximity is important because some of the points such as Input and Output should be placed only on the perimeter of the domain. In order to prevent the user from placing such points inside the volume of the cube, point proximity needs to be implemented.

e) The axis locking mechanism is used when the force vector needs to be along a specific coordinate axis. It corrects the direction of force vector and makes sure that there are no human errors while drawing it by free hand.

The two-dimensional design guidelines for the synthesis of Planar Compliant Mechanisms using load flow visualization method were validated by Krishnan et al. and Patiballa et al. [34, 35, 37]. These guidelines were also used to estimate the optimized stresses in design of PCM [50]. These guidelines were verified if they could be applied and extended to the SCM. The testing procedure involved the application of a simple SCM design on the VR software platform and verified if the expected design deformation occurred. The result of this procedure led to the replacement of two-dimensional design guidelines with that of the three dimensional. The initial stage of the research involved simple extension of two dimensional geometrical principles to that of three dimensional such as usage of hemispherical bands instead of semicircular ones. The key research finding was that a single constraint line was not sufficient for a single freedom ray at the intermediate point in a three-dimensional space. There was a need of at least two constraint lines representing a constraint plane. This observation was also applied for the constraints at the input and output points.

The VR software was finally validated through the application of five single input-single output compliant mechanism designs. All the five designs were found to be compatible with the three-dimensional design guidelines using load flow visualization method. This validation proved that the software could be extended to the application of other similar spatial compliant mechanisms. The key inferences of this validation include:

i) For the same design problem of a SCM, there can be multiple alternate solutions by changing the number of transmitters and/or intermediate points.

ii) The deflection direction of intermediate point depends on the orientation of the constraint plane with respect to the point.

5.2 FUTURE RECOMMENDATIONS FOR IMPROVEMENTS IN VR DESIGN SOFTWARE

a) The software algorithm can be extended from Virtual Reality to Augmented Reality for increasing the immersive experience. This software platform extension mainly requires change in build settings and minor editing of the source code. The ideal hardware for Augmented Reality can be Microsoft Hololens [52] or Magic Leap One [53]. The advantages of an Augmented Reality device for designing SCM include:

i) There will be no electrical wires and hence there is more freedom for the user to physically move around the augmented design domain.

ii) The user does not have to use any controllers [46] but use their own hands instead, to design the SCM in real three-dimensional physical space.

iii) The user can fix multiple compliant designs at different real physical locations and compare their deformation behaviors simultaneously. In the current VR software, after submitting the compliant design to Matlab [51] and receiving the analysis, the user again resets analysis, if the design turned out to be unsatisfactory. Instead we can fix a specific design at a physical location and place its alternate variations next to it in real physical space using Augmented Reality devices. This lets the user to qualitatively compare deformations of these designs simultaneously.

b) A ‘skip’ tool could be added for skipping multiple previous steps performed on the design domain and make desired changes to a specific prior step. This can be performed using a data entry text box again, where the user can mention a specific step number and directly skip to that step.

5.3 FUTURE RECOMMENDATIONS FOR APPLICATIONS OF VR SOFTWARE

a) The VR software can be customized for different design methods, such as PRBM and Topology Optimization, for spatial compliant mechanisms by performing essential modifications to the VR software features.

b) The VR software can also undergo necessary modifications to accommodate certain applications of Shape morphing and Tensegrity.

c) Patiballa et al. [50] developed load flow based designs for stress estimation. These designs can be extended to three-dimensional space with the help of Virtual Reality.

APPENDIX A –SOURCE CODE DESCRIPTION OF VR SOFTWARE

In this section, the source code of VR software is described in detail so that the users can customize the software according to their needs. The features of the key software components are flexible to undergo changes needed to accommodate any kind of spatial compliant mechanism. In order, to perform these changes to the software, the user needs to be educated about where a specific feature is executed in the source code. Hence, this section provides a detailed description of the source code generated for all the essential features of the software.

A.1 COORDINATES OF THE DOMAIN

```
18          Vector3 localCoord;
19          if (preview.activeSelf)
20          {
21              localCoord = domain.transform.InverseTransformPoint(preview.transform.position);
22          }
23          else
24          {
25              Vector3 pos = PointTypeSwitcher.GetComponent<PointTypeSwitcher>().GetPosition();
26              localCoord = domain.transform.InverseTransformPoint(pos);
27          }
28
29          localCoord.x += 0.5F;
30          localCoord.y += 0.5F;
31          localCoord.z += 0.5F;
32          localCoord.z = -localCoord.z;
33          localCoord.z += 1.0F;
34          coordText.text = localCoord.ToString("F4");
```

Figure A.1: Script for assigning coordinate values to different point of design domain

The localcoord variable is assigned with a global position vector using Vector3 function in Line 18. The if statement from Lines 19-22 ensures that when the Preview Gameobject is set on active mode, local coordinates of the cube is set with respect to the origin of the domain space. It is obtained by converting the global position of the preview gameobject into the local position with respect to the domain, using Transform.InverseTransformPoint function. The preview Gameobject helps the user to visually see the coordinates (X, Y, Z) of cursor, as the user moves the cursor over the volume of the cube domain. This is a very useful feature when the user knows the coordinates

of a specific point and desires to place it on the domain. The else function (Lines 23-27) is implemented when the point is randomly placed on the cube domain and the user would like to assign coordinates to the point. The global position of the PointTypeSwitcher Gameobject is captured in pos variable and this is converted into local position with respect to the cube domain. This coordinate values is assigned to the localcoord variable (Figure A.1).

Due to the offset in the global position of the Preview and PointTypeSwitcher, there was a need for recalibration of the local coordinates of a point within the domain space. This was being executed in Lines 29-33. These coordinates are transferred to a Textbox over the right controller [46] and it gets updated in real time, as the user moves the cursor around the domain space. This is performed using ToString function in Line 34 (Figure A.1).

A.1.1 RESIZE OF THE DOMAIN

KEY VARIABLES IN FIGURES x and y:

initDist – Initial Distance between the two Touch Controllers [46]; currDist – Current Distance (After Scaling the cube) between the two Touch Controllers [46]; initScale – Initial Scale of the Cube Domain; currScale – Current Scale (After Scaling the cube) of the Cube Domain; totalMagnification – Ratio of Current Scale to Initial Scale; Lcontroller_loc – Local position of Left Touch Controller [46]; Rcontroller_loc – Local position of Right Touch Controller [46].

```
10          private float initDist, currDist = -1;
11          private Vector3 initScale;
12          private Vector3 currScale;
13          private float totalMagnification;
14          void Start()
15          {
16              initScale = gameObject.transform.localScale;
17              totalMagnification = 1;
18          }
19          // Update is called once per frame
20          void Update()
21          {
22              if (OVRInput.Get(OVRInput.Button.PrimaryIndexTrigger) && OVRInput.Get(OVRInput.Button.SecondaryIndexTrigger))
23              {
24                  Vector3 Lcontroller_loc = OVRInput.GetLocalControllerPosition(LController);
25                  Vector3 Rcontroller_loc = OVRInput.GetLocalControllerPosition(RController);
26                  if (initDist == -1 && currDist == -1 || initDist == 0 && currDist == -1)
27                  {
28                      initDist = Vector3.Distance(Lcontroller_loc, Rcontroller_loc);
29                      currScale = gameObject.transform.localScale;
30                  }
31                  else
32                  {
33                      currDist = Vector3.Distance(Lcontroller_loc, Rcontroller_loc);
34                      gameObject.transform.localScale = currScale * currDist / initDist;
35                      //print("initDist: " + initDist);
36                      //print("currDist: " + currDist);
37                  }
38
39              }
```

Figure A.2: Script to apply distance conditions to scale the design domain

In Lines 14-18, the initial scale of the cube domain is assigned to initScale variable and totalMagnification variable is set to 1, since the cube is initially in its default size at the beginning of the game. These lines are only executed in the beginning of the game i.e. first second, since they fall under Start() built-in function. Lines 20-39 are executed every frame in a loop after the game has started, since they fall under Update() built-in function. The if statement, in Line 22, is to apply a condition if the squeeze triggers of both left and right controllers [46] are pressed. If the condition in Line 22 becomes true, then the local positions of both left and right controllers [46] are captured using GetLocalControllerPosition() built-in function. The left and right squeeze controllers [46] are referred to as PrimaryIndexTrigger and SecondaryIndexTrigger respectively in terms of control input for Get() sub-function under OVRInput() built-in function. Then, another simultaneous condition is checked using a nested if statement. The condition is to check if initial distance and current distance have remained constant. If it turns out to be true, then the lines 28-29 requests Unity [43] to keep the scale of the domain constant. So, the purpose of this nested if statement is to make sure that the scale does not get affected as long as the distance between the left and right controllers [46] is constant. If the above nested if statement turns out to be False, then else

statement from Lines 31-37 will be executed. In Line 34, the cube domain would be magnified according to the ratio of currDist to initDist, since this ratio is being multiplied to the currScale. This else statement is crucial because the user need not spread the left and right controllers [46] away from each other even after pressing the squeeze triggers and this statement makes sure distance is increased/decreased to scale the domain, while they are pressed (Figure A.2).

```
40          if (OVRInput.GetUp(OVRInput.Button.PrimaryIndexTrigger) || OVRInput.GetUp(OVRInput.Button.SecondaryIndexTrigger))
41          {
42              initDist = -1;
43              currDist = -1;
44
45              totalMagnification = gameObject.transform.localScale.x / initScale.x;
46              if (totalMagnification >= 8)
47              {
48                  gameObject.transform.localScale = initScale * 8;
49                  totalMagnification = 8;
50              }
51              else if (totalMagnification <= 0.05)
52              {
53                  gameObject.transform.localScale = initScale * 0.05F;
54                  totalMagnification = 0.05F;
55              }
56
57
58          }
59
60      }
61
62      public float getMagnification()
63      {
64          return totalMagnification;
65      }
66
```

Figure A.3: Script for setting maximum and minimum scaling ratio at the end of resizing execution

In Line 40, the if statement is applied if the squeeze triggers of either left or right controllers [46] are released by the user. The upper limits of the scaling is set to 8 within the if statement in Lines 46-50, while the lower limits is set to 0.5 within the elseif statement in Lines 51-55. Hence, the script in Figure 14, makes sure that the cube domain is locked down to the scaled magnitude set by the user, once they release one or both the squeeze triggers (Figure A.3).

A.2 DATA EXTRACTION OF INPUT AND OUTPUT POINTS

```
5   public class InputOutputInfo : MonoBehaviour {
6
7       private GameObject origin;
8       private GameObject force;
9       private GameObject hemisphere;
10
11      private Vector3 forceVector;
12
13      private List<GameObject> connectedPoints;
14      private bool directionTowardsOrigin;
15
16      private bool viability;
17
```

Figure A.4: Script for initializing key variables for capturing input-output information

The gameobjects origin, force and hemisphere are defined private since they are applicable only within this specific script/class (Figure A.4). In addition, there are variables of type point Vector3 (forceVector), List (connectedPoints), Boolean (directionTowardsOrigin and viability).

```
18      void Start()
19      {
20          connectedPoints = new List<GameObject>();
21      }
22      public bool Setup(GameObject originPoint, GameObject forcePoint, GameObject hemisphereObj, Vector3 forceVector, bool directionTowardsOrigin)
23      {
24          this.origin = originPoint;
25          this.force = forcePoint;
26          this.hemisphere = hemisphereObj;
27          this.forceVector = forceVector;
28          this.directionTowardsOrigin = directionTowardsOrigin;
29
30          return checkFeasibility();
31      }
32
33      public void SetOriginPoint(GameObject originPoint)
34      {
35          this.origin = originPoint;
36      }
37
38      public void SetForcePoint(GameObject forcePoint)
39      {
40          this.force = forcePoint;
41      }
42
```

Figure A.5: Script for assigning initialized variables to input arguments of various functions

Under the Start() function, a new list of gameobjects is created [Lines 18-21]. Setup() function takes the input of all the variables initialized previously and returns Boolean output [Line 22]. The keyword ‘this’ refers to the object of this script class ‘InputOutputInfo’. All the initialized variables are assigned to the respective input arguments of the Setup() function [Lines 24-28].

Finally Setup() function returns checkFeasibility() function (defined later in this script) [Line 30] (Figure A.5).

Similarly, SetOriginPoint() and SetForcePoint() functions have their input arguments (originPoint and forcePoint) assigned to the initialized variables (origin and force) of this script (Figure A.5).

```
43          public void DeleteForcePoint()
44          {
45              this.force = null;
46              this.forceVector = new Vector3(-1, -1, -1);
47              this.hemisphere = null;
48          }
49
50          public bool SetForceVector(Vector3 forceVector)
51          {
52              this.forceVector = forceVector;
53              return checkFeasibility();
54
55          }
56          public GameObject GetOriginPoint()
57          {
58              return origin;
59          }
60
61          public GameObject GetForcePoint()
62          {
63              return force;
64          }
65
66          public Vector3 GetForceVector()
67          {
68              return forceVector;
69          }
70
71          public GameObject GetHemisphereObj()
72          {
73              return hemisphere;
74          }
75
76          public bool GetDirection()
77          {
78              return directionTowardsOrigin;
79          }
```

Figure A.6: Script for introducing various functions for getting key design elements

In DeleteForcePoint() function, the force and hemisphere variables are not assigned to anything. While the forceVector variable is assigned with position coordinates (-1, -1, -1) [Lines 43-48]. The SetForceVector() function returns the checkFeasibility() function and also assigns the initialized variables similar to the previous functions. All the functions namely GetOriginPoint(), GetForcePoint(), GetForceVector(), GetHemisphereObj(), GetDirection() return the respective

initialized variables namely origin, force, forceVector, hemisphere, directionTowardsOrigin. It is worthy to note the type of these functions return. For example, GetForcePoint() function returns gameobject type (Figure A.6).

```
110         public void toggleHemisphereView(bool visible)
111         {
112             if(hemisphere != null)
113                 hemisphere.GetComponent<MeshRenderer>().enabled = visible;
114         }
115
116         public void addConnection(GameObject vertex)
117         {
118             connectedPoints.Add(vertex);
119         }
120
121         public void removeConnection(GameObject vertex)
122         {
123             connectedPoints.Remove(vertex);
124         }
125
126         private bool checkViability(Vector3 forceVector, Vector3 vertexVector)
127         {
128             if (Vector3.Angle(forceVector, vertexVector) > 90)
129                 return false;
130             else
131                 return true;
132         }
```

Figure A.7: Script for various functions executing different actions on design elements

The toggleHemisphereView() function takes in a Boolean input argument ‘visible’ and does not return anything [Line 110]. If the hemisphere gameobject is called, then its MeshRenderer Component is enabled/activated and it is assigned to ‘visible’ variable [Lines 112-114]. The addConnection() and removeConnection() variables are used to add and remove the gameobject ‘vertex’ to the ‘connectedPoints’ list [Lines 116-124]. The earlier mentioned function checkViability() takes in the input arguments namely ‘forceVector’ and ‘vertexVector’ respectively [Line 126]. The if condition checks if the angle between forceVector and vertexVector variables is obtuse and if it turns out to be true, then it returns checkViability function as false [Lines 128-129]. Otherwise, it returns the function to be true [Line 131] (Figure A.7).

A.3 DATA EXTRACTION OF INTERMEDIATE POINT

```
1     using System.Collections;
2     using System.Collections.Generic;
3     using UnityEngine;
4     using Vectrosity;
```

Figure A.8: Script for importing key libraries needed for capturing Intermediate points

This script has an additional library called Vectrosity [45] and it imports the Vectrosity [45] asset (Figure A.8).

```
7     public class IntermediateInfo : MonoBehaviour {
8
9         private GameObject intermediatePoint;
10
11        private List<GameObject> connectionList;
12
13        private List<GameObject> hemisphereList;
14
15        private GameObject truncatedHemisphere;
16
17        private Material hemisphereMaterial1;
18        private Material truncatedHemisphereMaterial;
19        private Material planeMaterial;
20
21        private bool hemisphereVisible;
22        private bool truncationExists;
23
24        private bool viability;
25
26        private GameObject domain;
27
28        private GameObject plane;
29        private GameObject freedomLineVectorObj;
30        private VectorLine freedomLineVector;
```

Figure A.9: Script for initializing key variables needed for capturing Intermediate points

Various variables are initialized and their types include gameobject, list, material, Boolean and Vectorline [49] (datatype within Vectrosity [45] asset) [Lines 9-30]. The key note is that the list variable comprises of elements of type gameobject (Figure A.9).

```
32          public void SetupMaterials(Material hemisphereMat1, Material truncatedHemisphereMat, Material planeMat)
33          {
34              this.hemisphereMaterial1 = hemisphereMat1;
35              this.truncatedHemisphereMaterial = truncatedHemisphereMat;
36              this.planeMaterial = planeMat;
37          }
38
39          public void SetObjs(GameObject intermediatePoint, GameObject domain)
40          {
41              connectionList = new List<GameObject>();
42              hemisphereList = new List<GameObject>();
43              this.intermediatePoint = intermediatePoint;
44              this.domain = domain;
45              freedomLineVectorObj = null;
46              freedomLineVector = null;
47              hemisphereVisible = true;
48              truncationExists = false;
49
50              plane = null;
51          }
52
```

Figure A.10: Script for assigning initialized variables to input arguments of various functions

The SetupMaterials() function takes in various input arguments of type Material [Line 32] and returns nothing. This function assigns the initialized material variables to its input arguments [Lines 34-36]. Similarly, the SetObjs() function takes in input arguments of type Gameobject and return nothing [Line 39]. In this function, the initialized list variables are assigned with empty list [Lines 41-42] and few initialized variables are assigned to the respective input arguments of the function [Lines 43-44]. One of the gameobject and Vectorline type initialized variables are given null value [Lines 45-46 and 50] and some of the Boolean variables are assigned with some Boolean value [Lines 47-48] (Figure A.10).

```
53          public List<GameObject> GetConnections()
54          {
55              return connectionList;
56          }
57
58          public List<GameObject> GetHemispheres()
59          {
60              return hemisphereList;
61          }
62
63          public VectorLine GetFreedomLine()
64          {
65              return freedomLineVector;
66          }
67
68          public GameObject GetFreedomLinePObj()
69          {
70              return freedomLineVectorObj;
71          }
72
```

Figure A.11: Script for receiving information about key design elements

The functions GetConnections() and GetHemispheres() return List type having the elements of type Gameobject [Lines 53-61]. While GetFreedomLine() and GetFreedomLinePObj() functions return VectorLine and Gameobject type respectively [Lines 63-71]. All these functions return the respective initialized variables (Figure A.11).

```
73          public float GetScaleAsFloat()
74          {
75              if (truncationExists)
76              {
77                  return hemisphereList[0].transform.lossyScale.x;
78              }
79              else
80                  return 0;
81          }
82
83          public Vector3 GetScale()
84          {
85              if (truncationExists)
86              {
87                  return hemisphereList[0].transform.lossyScale;
88              }
89              else
90                  return new Vector3(0, 0, 0);
91          }
92
93          public GameObject GetPlane()
94          {
95              return plane;
96          }
97
98
```

Figure A.12: Script for actions to be executed, if truncation exists

The GetScaleAsFloat() function returns float type and checks if 'truncationExists' is true. If it is true, then it returns the lossyScale of the first element of the list 'hemisphereList'. The lossyScale is of Vector3 type and it returns the global/world scale of that object. On the other hand, when the if condition turns false, it returns value zero to the GetScaleAsFloat() function [Lines 73-81] (Figure A.12).

The GetScale() function is similar to that of GetScaleAsFloat() function except for the following reasons. GetScale() function returns Vector3 datatype and when the if condition turns false, it returns a point to the function [Lines 83-91]. The GetPlane() returns gameobject type and it can be observed that it returns one of the initialized variable [Lines 93-96] (Figure A.12).

A.4 COLOR FOR INTERMEDIATE POINT AND TRUNCATED HEMISPHERE

```
288         public void HighlightPoint(bool highlight)
289         {
290             if (!hemisphereVisible)
291                 return;
292             float alpha = 0;
293             if (highlight)
294                 alpha = 1;
295             else
296                 alpha = 0.353F;
297 
298 
299             Color color = intermediatePoint.GetComponent<Renderer>().material.color;
300             color.a = alpha;
301             intermediatePoint.GetComponent<Renderer>().material.color = color;
302 
303             if(truncationExists)
304             {
305                 color = truncatedHemisphere.GetComponent<Renderer>().material.color;
306                 color.a = alpha;
307                 truncatedHemisphere.GetComponent<Renderer>().material.color = color;
308             }
309         }
310     }
311 
```

Figure A.13: Script for setting same color for the truncated hemisphere and intermediate point

The HighlightPoint() function takes in the input arguments of type boolean and returns nothing [Line 288]. The first if condition checks if the hemisphere is not visible and in that case,

it returns nothing [Lines 290-291]. The second if condition checks if the point is highlighted and in that case alpha component of color is set to 1; otherwise it is set to 0.353 [Lines 293-296]. The 'color' variable is assigned to the color of the 'intermediatePoint' gameobject. This is obtained from the material sub-component within the Renderer component of the 'intermediatePoint' gameobject [Line 299]. Then the alpha component of this 'color' variable is assigned to the 'alpha' variable [Line 300]. Then this modified color is assigned back to the intermediatePoint's color [Line 301]. The final if statement checks if the truncation exists and if so, then the 'color' variable is assigned to the color of the 'truncatedHemisphere' gameobject. This is obtained from the material sub-component within the Renderer component of the 'truncatedHemisphere' gameobject [Line 305]. Then the alpha component of this 'color' variable is assigned to the 'alpha' variable [Line 306]. Then this modified color is assigned back to the truncated Hemisphere's color [Line 307] (Figure A.13).

A.5 PROPERTIES OF FREEDOM LINE AND CONSTRAINT PLANE

```
 99          public void SetFreedomLine(VectorLine freedomLine, GameObject freedomLineObj)
100          {
101              this.freedomLineVector = freedomLine;
102              this.freedomLineVectorObj = freedomLineObj;
103          }
104
105          public void removeFreedomLineAndPlane()
106          {
107              VectorLine.Destroy(ref freedomLineVector);
108              GameObject.Destroy(plane);
109
110              freedomLineVector = null;
111              freedomLineVectorObj = null;
112              plane = null;
113          }
```

Figure A.14: Script for creating and removing freedom line and plane

The SetFreedomLine() function takes in input arguments of types VectorLine and Gameobject and returns nothing [Line 99]. Some of the initialized variables are assigned to the input arguments of the function [Lines 101-102]. In removeFreedomLineAndPlane() function, the Destroy function is used to destroy the Vectorline 'freedomLineVector' and Gameobject 'plane'

respectively [Lines 107-108]. This function also has few initialized variables set to null [Lines 110-112] (Figure A.14).

```
277         public void SpawnPlane(GameObject target, Vector3 size)
278         {
279             plane = GameObject.CreatePrimitive(PrimitiveType.Cube);
280             //new Vector3(10, 10, 0.001F)
281             plane.transform.localScale = size;
282             plane.transform.rotation = Quaternion.LookRotation(target.transform.position - intermediatePoint.transform.position);
283             plane.transform.position = intermediatePoint.transform.position;
284             plane.GetComponent<MeshRenderer>().sharedMaterial = planeMaterial;
285             plane.transform.parent = intermediatePoint.transform;
286         }
287
```

Figure A.15: Script for creating a constraint plane and setting its dimensions

The SpawnPlane() function takes in the input arguments of type Gameobject and Vector3 and returns nothing [Line 277]. The gameobject ‘plane’ is created in the shape of a cube with a primitive mesh renderer and appropriate collider, using CreatePrimitive() function [Line 279]. Then the localScale and position of the plane are assigned to various initialized variables. Also, orientation of the plane is along the forward direction of the distance between target and intermediatePoint [Lines 281-283]. The material of the ‘plane’ gameobject is assigned to ‘planeMaterial’ variable [Line 284]. The ‘intermediatePoint’ gameobject is assigned as the parent to the ‘plane’ gameobject (Figure A.15).

A.6 SUBMIT DESIGN TOOL AND CONNECTION BETWEEN UNITY AND MATLAB

```
1   using UnityEngine;
2   using System.Collections;
3   using System.Net;
4   using System.Net.Sockets;
5   using System.Linq;
6   using System;
7   using System.IO;
8   using System.Text;
9   public class DataTransfer : MonoBehaviour
10  {
11      // Use this for initialization
12      TcpListener listener;
13      StreamWriter theWriter;
14      String msg;
15      void Start()
16      {
17          listener = new TcpListener(55001);
18          listener.Start();
19          print("is listening");
20      }
21
```

Figure A.16: Script for establishing the socket connection between Matlab [51] and Unity [43]

The integration of Matlab [51] and Unity [43] requires important libraries such as System.Net and System.Net.Sockets. TcpListener function refers to the receiver of data between Matlab [51] and Unity [43]. For instance, if Matlab [51] receives data from Unity [43], then Matlab [51] is referred to as listener variable. In the above instance, Unity [43] is theWriter variable and the function StreamWriter is applied on it. The data transferred is referred to as msg variable. The port number used for establishing the connection is 55001. The function listener.Start() activates the receiver end i.e. Matlab [51] and it would be ready to receive the data (Lines 1-20) (Figure A.16).

```
22          void Update()
23          {
24              if (testConnection())
25              {
26                  SendData("Hello");
27              }
28          }
29          public bool testConnection()
30          {
31              if (!listener.Pending())
32              {
33                  return false;
34              }
35              return true;
36          }
37          public void SendData(string data)
38          {
39
40              print("socket comes");
41              TcpClient client = listener.AcceptTcpClient();
42              NetworkStream ns = client.GetStream();
43              StreamReader reader = new StreamReader(ns);
44              theWriter = new StreamWriter(ns);
45              theWriter.AutoFlush = true;
46              theWriter.WriteLine(data);
47              Debug.Log("socket is sent");
48          }
49
```

Figure A.17: Script for connection testing and data transfer

The connection is requested by the listener.AcceptTcpClient() function. The data stream is received, read and written from the Client server (Unity [43]) by using GetStream(); StreamReader(); StreamWriter() functions (Lines 41-47). The if-else statements, in Lines 24-36, test the connection and verify if the client server and listener servers are sending and receiving the data successfully (Figure A.17).

A.7 PROPERTIES OF RESET ALL

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement;
5   public class ResetAll : MonoBehaviour
6   {
7       /*
8        * This class is used solely for the Reset All Button and is used to reset the scene.
9        */
10      public void ResetAllButton()
11      {
12          SceneManager.LoadScene(SceneManager.GetActiveScene().name);
13      }
14  }
```

Figure A.18: Script for refreshing the design domain to default setting

In order to reload a specific Unity [43] scene, the key library to be accessed is the UnityEngine.SceneManagement [48] (Line 4). The crucial line of script is that of Line 14, since it has multiple mechanisms going on. Initially, the SceneManager recovers the currently active scene by its name and that specific scene is loaded [48]. This eventually leads to reloading the game back to the beginning of empty domain. This SceneManagement has been introduced recently after Unity [43] 5.3 versions and thus reducing the need to write multiple lines of script [48] (Figure A.18).

A.8 PROPERTIES OF CANCEL TOOL

```
12          private GameObject originSphere;
13          private Vector3 origin;
14          private Vector3 dest;
15          private bool originSet;
16          private bool isColliding;
17          private GameObject currCollidingObj;
18          private float gridGranularity;
19          private Vector3 closestPoint;
20          private bool allowPlacing;
21          private VectorLine delLine;
22
23          [Tooltip("GameObject of the Domain Cube")]
24          public GameObject domain;
25          [Tooltip("GameObject of Right Controller")]
26          public GameObject RightController;
27          [Tooltip("GameObject of Preview Sphere")]
28          public GameObject preview;
29          [Tooltip("GameObject of Networking")]
30          public GameObject Networking;
31          [Tooltip("GameObject of CenterEyeAnchor")]
32          public GameObject myCamera;
```

Figure A.19: Script for initialization of key variables for Cancel tool

The Cancel tool is universally used on various components of the design software such as points, lines, force vector etc. Hence, it requires lots of variables and sub-functions that execute various unique operations, in order to remove these components. The variables initialized in Lines 12-21 include originSphere (Position of first deleting sphere that is placed for deleting a line); dest (position of destination sphere that is placed for deleting a line); isColliding (verifies if the deleting sphere coincides with the point to be deleted); originSet (verifies if the originSphere is designated); currCollidingObj (set to the point currently colliding with the preview); gridGranularity (Grid lines forming a mesh inside the cube); closestPoint (coordinates to the closest point on the grid system to the sphere on the right controller [46]); allowPlacing (indicates whether the sphere is allowed to be placed); delLine (preview line used to show lines that will be deleted). The data types for various variables include bool (indicate true/false); Vector3 (Position Coordinates); Vectorline (3D ray drawn by the user) (Figure A.19). Here Tooltip class is used for providing Description Notes on a specific public variable defined in the Inspector of Unity Editor [43] (as shown in Figure A.20).

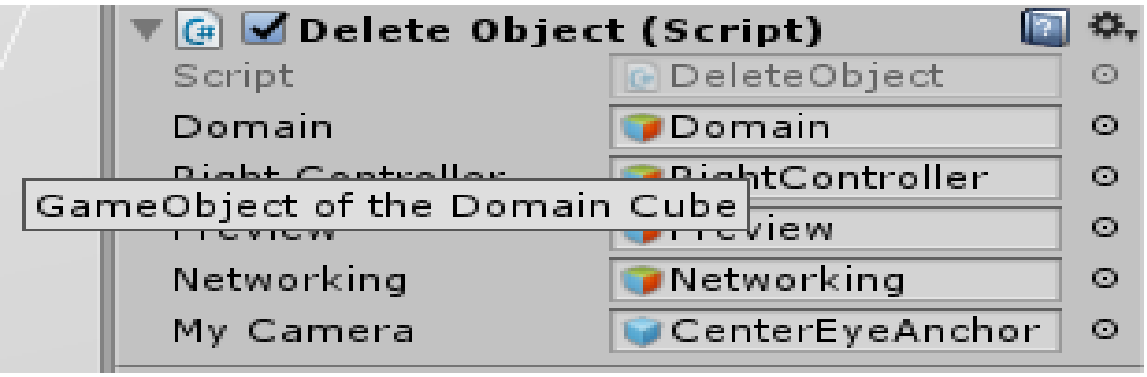


Figure A.20: Illustration of Descriptive Text "Gameobject of the Domain Cube" over variable 'Domain' in the Inspector of Unity Editor [43] (With the permission of Unity Technologies. All rights reserved.)

```
116         void getClosestPoint()
117         {
118             float tileSize = domain.transform.localScale.x * gridGranularity;
119             Vector3 vectorToLoc = gameObject.transform.position - domain.transform.position;
120             vectorToLoc = domain.transform.InverseTransformDirection(vectorToLoc);
121             Vector3 relativePos = new Vector3();
122             relativePos.x = Mathf.Round(vectorToLoc.x / tileSize) * tileSize;
123             relativePos.y = Mathf.Round(vectorToLoc.y / tileSize) * tileSize;
124             relativePos.z = Mathf.Round(vectorToLoc.z / tileSize) * tileSize;
125
126             relativePos = domain.transform.TransformDirection(relativePos);
127             closestPoint = relativePos + domain.transform.position;
128         }
```

Figure A.21: Script for sub-function of getting closest desired point to the preview cancel point

According to triangulation law of vectors, the relative position of destination sphere is the subtraction of domain in global space from its position vector in the global space. This is referred to as vectortoLoc variable. The tile size is the product of the scale of the domain and grid granularity (i.e. the factor of breaking down the domain space into grid cells). The relativePos variable is derived by rounding off the ratio between vectortoLoc to TileSize. Then, this ratio is multiplied to the TileSize of the domain, which results in relativePos variable in domain space. Then relativePos variable is converted from local space to global space. Finally, the triangulation rule of vectors is applied to derive the closest point distance from relative position and domain's position in global space. This sub-function getClosestPoint() is essential since the software needs to detect the closest points to the user, rather than selecting farther away unwanted points behind them (Figure A.21).

A.9 CREATION AND TRUNCATION OF HEMISPHERES

```
114         public bool addConnection(GameObject connection, bool drawnTowardConnection)
115         {
116             if (connectionList.Contains(connection))
117                 return false;
118             if (connection.CompareTag("Intermediate"))
119                 return false;
120 
121             //first, generate a hemisphere
122             bool dirTowardConnection = false;
123             if(connection.CompareTag("Input") || connection.CompareTag("Output"))
124             {
125                 dirTowardConnection = calculateForceDirection(this.intermediatePoint, connection);
126             }
127             else
128             {
129                 dirTowardConnection = drawnTowardConnection;
130             }
131 
132             Vector3 scale = new Vector3(5, 5, 5);
133             GameObject hemisphere = Hemisphere.CreateHemisphere(hemisphereMaterial1, intermediatePoint.transform.position,
134                                     connection.transform.position,dirTowardConnection, scale);
135             connectionList.Add(connection);
136             hemisphereList.Add(hemisphere);
137             hemisphere.transform.parent = intermediatePoint.transform;
138             hemisphere.transform.localScale = scale;
139             truncatedHemisphere = truncate(hemisphere, truncatedHemisphere);
140             return true;
141 
142         }
143 
```

Figure A.22: Script for adding connection and creating a hemisphere

The addConnection() function takes the input arguments of type Gameobject and Boolean and returns Boolean value [Line 114]. The first if condition checks if the list ‘connectionList’ has the element ‘connection’ using the Contains() function and if it turns out to be true, then the function returns false [Lines 116-117]. The second if condition checks if the ‘connection’ variable has the tag name ‘Intermediate’ and if it turns out to be true, then the function returns false [Lines 118-119]. The third if-else statement checks if the ‘connection’ variable has the tag name ‘Input’ or ‘Output’ and if it turns out to be true, then the variable ‘dirTowardConnection’ will be assigned to the function ‘calculateForceDirection’. Otherwise, it will assign the variable ‘dirTowardConnection’ to the input argument of the function [Lines 123-130] (Figure A.22).

The hemisphere gameobject is assigned to the CreateHemisphere() function within the Hemisphere script. The initialized variables are assigned to the respective input arguments of

CreateHemisphere() function [Lines 133-134]. Some of the initialized variables are added to some of the other initialized list variables, using Add() function [Lines 135-136]. The 'intermediatePoint' variable is assigned as parent to the 'hemisphere' variable [Line 137]. The 'scale' variable is assigned with the localScale of the hemisphere. The local scale refers to the scale of a child relative to its parent [Line 138]. The 'truncatedHemisphere' variable is assigned to the function 'truncate' and if none of the above if statements doesn't work, then the 'addConnection' function returns 'true' value [Lines 139-140] (Figure A.22).

```
144         public bool removeConnection(GameObject connection)
145         {
146             if (!connectionList.Contains(connection))
147             {
148                 return false;
149             }
150             else
151             {
152                 int connectionIndex = connectionList.IndexOf(connection);
153                 Destroy(truncatedHemisphere);
154                 Destroy(hemisphereList[connectionIndex]);
155                 connectionList.RemoveAt(connectionIndex);
156                 hemisphereList.RemoveAt(connectionIndex);
157                 recalculateTruncation();
158                 return true;
159             }
160         }
161
162         public void toggleHemisphereView(bool visible)
163         {
164             if (truncationExists)
165             {
166                 hemisphereVisible = visible;
167                 truncatedHemisphere.GetComponent<MeshRenderer>().enabled = visible;
168                 if(freedomLineVectorObj != null)
169                     freedomLineVectorObj.SetActive(visible);
170             }
171         }
172
```

Figure A.23: Script for removing connection and toggling Hemisphere

The removeConnection() function takes in the input argument of type Gameobject and returns Boolean value [Line 144]. The if-else statement checks if the list 'connectionList' contains 'connection' element and if the condition is false, then removeConnection() function returns false [Lines 146-149]. Otherwise, the function returns true and also destroys the 'truncatedHemisphere' variable and an element of list 'hemisphereList' at the index of 'connectionIndex' [Lines 153-154]. The RemoveAt() finction is used to remove a specific element at the index of

‘connectionIndex’ at the respective list variables [Lines 155-156]. The ‘connectionIndex’ variable is index of connection variable, using IndexOf() function [Line 152]. In addition, the recalculateTruncation() function is also called [Line 157] (Figure A.23).

The toggleHemisphereView() function takes in the input argument of type boolean and returns nothing [Line 162]. The nested if statement checks if the truncationExists is true and if it is so, then ‘truncatedHemisphere’ gameobject’s MeshRenderer Component is enabled/activated and it is assigned to ‘visible’ Boolean variable [Lines 166-167]. The second if statement checks if the gameobject ‘freedomLineVectorObj’ is true and in that case, it is activated [Lines 168-169] (Figure A.23).

```
175         private GameObject truncate(GameObject newHemisphere, GameObject existingTruncation)
176         {
177             if(newHemisphere == null)
178             {
179                 return null;
180             }
181             if (connectionList.Count == 1) //this is the only connection, nothing to do
182             {
183                 truncationExists = true;
184                 viability = false;
185                 return newHemisphere;
186             }
187             float scaling = 1;
188             if (domain.GetComponent<ResizeObject>().getMagnification() >= 1)
189             {
190                 scaling = 10;
191             }
192             else
193             {
194                 scaling = 20;
195             }
196             domain.transform.localScale = domain.transform.localScale * scaling;
197             GameObject truncation = Hemisphere.GetIntersection(existingTruncation, newHemisphere, truncatedHemisphereMaterial, true);
198 
199             if (truncation == null) //means there was no ove rlapping region
200             {
201                 viability = false;
202                 truncationExists = false;
203                 domain.transform.localScale = domain.transform.localScale / scaling;
204                 return truncation;
205             }
206             else
207             {
208                 viability = true;
209                 truncationExists = true;
210                 truncation.transform.parent = newHemisphere.transform.parent;
211             }
212 
213             domain.transform.localScale = domain.transform.localScale / scaling;
214             return truncation;
215         }
216 
```

Figure A.24: Script for checking the truncation of input and output hemispheres

The truncate() function takes in the input arguments of type Gameobject and returns gameobject type [Line 175]. The first if statement checks for the absence of a variable and returns the function as null [Lines 177-180]. In the condition of second if statement, Count() function is used to count the number of elements within the list variable ‘connectionList’ [Line 181]. If the number of elements is 1, then various initialized Boolean variables are given corresponding Boolean values and the truncate() function returns one of the input argument [Lines 183-186] (Figure A.24).

The third if-else statement checks if the value of ‘totalMagnification’ variable is greater than or equal to 1. This is done by calling the getMagnification() function from ‘ResizeObject’ script attached to ‘domain’ gameobject. If it turns out to be true, then a specific value is given to ‘scaling’ variable or otherwise some other value will be allotted [Lines 188-195]. Then, the localScale of domain gameobject will be magnified according to the value of ‘scaling’ variable [Line 196]. The gameobject ‘truncation’ is assigned to ‘GetIntersection’ function of Hemisphere script [Line 197] (Figure A.24).

The final if-else statement checks if there is overlap/intersection between the hemispheres. If there is no overlap, various initialized variables are given ‘false’ Boolean value and localScale of the ‘domain’ gameobject is shrunk and shows no ‘truncation’ gameobject [Lines 199-205]. Otherwise, the above initialized variables are given ‘true’ Boolean value and ‘truncation’ and ‘newHemisphere’ gameobjects become children of the same parent gameobject [Lines 206-211]. Finally, the ‘domain’ gameobject will be shrunk and ‘truncation’ gameobject will be returned by the truncate() function [Lines 213-214] (Figure A.24).

```
217         private void recalculateTruncation()
218         {
219             if(connectionList.Count == 0)
220             {
221                 truncatedHemisphere = null;
222                 viability = false;
223                 truncationExists = false;
224             }
225             else if(connectionList.Count == 1)
226             {
227                 truncatedHemisphere = hemisphereList[0];
228                 truncatedHemisphere.GetComponent<MeshRenderer>().enabled = true;
229                 viability = false;
230                 truncationExists = true;
231             }
232             else
233             {
234                 GameObject truncation = hemisphereList[0];
235                 for(int i = 1; i < hemisphereList.Count; i++)
236                 {
237                     truncation = truncate(hemisphereList[i], truncation);
238                 }
239                 truncatedHemisphere = truncation;
240                 truncatedHemisphere.GetComponent<MeshRenderer>().enabled = true;
241             }
242         }
```

Figure A.25: Script for recalculating truncation of input and output hemispheres

The recalculateTruncation() function checks the number of elements in list variable ‘connectionList’ and applies the applicable execution statements accordingly. If the number of elements is zero, then it assigns ‘false’ value to the initialized variables [Lines 219-224]. While if it is one, then ‘truncatedHemisphere’ variable is assigned to the first element of list variable ‘hemisphereList’. Also, various initialized Boolean variables are given various Boolean values [Lines 225-231]. If the number of elements are more than one, then the truncation gameobject will be assigned to the first element of list variable ‘hemisphereList’. Also, the for loop will be executed on the truncate() function for each element of list variable ‘hemisphereList’. Finally, ‘truncation’ gameobject will be assigned to ‘truncatedHemisphere’ gamobject [Lines 232-241] (Figure A.25).

A.10 TOGGLE VISIBILITY OF HEMISPHERES

```
47      public void toggleHemisphereView(bool visible)
48      {
49          if (hemisphere != null)
50              hemisphere.GetComponent<MeshRenderer>().enabled = visible;
51      }

54      public void toggleHemisphereView(bool visible)
55      {
56          if (truncationExists)
57          {
58              hemisphereVisible = visible;
59              truncatedHemisphere.GetComponent<MeshRenderer>().enabled = visible;
60              if (freedomLineVectorObj != null)
61                  freedomLineVectorObj.SetActive(visible);
62          }
63      }
```

Figure A.26: Script for toggling view of partial and complete hemispheres

The subfunction toggleHemisphereView is used for both I/O points as well as intermediate points. The if statement, in Lines 48-51, renders meshes to form the hemisphere, once it is initialized through placing a point by the user. While the if statements in Lines 56-62 are almost similar to the previous one, the difference is that there is a nested if applied on the freedom line vector and its initialization. This is made to ensure that the freedom line vector is kept visible after it is being drawn by the user and continued to be visible even after the design analysis (Figure A.26).

```
15          void Start()
16          {
17              visible = true;
18          }
19          void Update()
20          {
21
22              foreach (Transform transform in domain.transform)
23              {
24                  if (transform.gameObject.CompareTag("Input") || transform.gameObject.CompareTag("Output"))
25                  {
26                      transform.gameObject.GetComponent<InputOutputInfo>().toggleHemisphereView(visible);
27                  }
28                  if (transform.gameObject.CompareTag("Intermediate"))
29                  {
30                      transform.gameObject.GetComponent<IntermediateInfo>().toggleHemisphereView(visible);
31                  }
32              }
33              if (OVRInput.GetDown(OVRInput.RawButton.X))
34              {
35                  if (visible)
36                  {
37                      visible = false;
38                  }
39                  else
40                  {
41                      visible = true;
42                  }
43              }
44          }
```

Figure A.27: Script for controller [46] inputs to carry out the toggling operation

In this main script, initially, the hemispheres are made active and visible, since that is the desired outcome (Lines 15-17). The if statements, in Lines 24 to 31, convey that if the gameobject have a tag/label as either Input or Output, then the script for I/O should be activated and the hemisphere should show up at either of these points. Similarly, it applies the same to even intermediate point. The if statement at the end of this script (Lines 33-42) mentions that the toggle action will be applied by pressing button ‘X’ on the left controller [46]. Hence, if the hemisphere was previously visible, then it can be made invisible by pressing button ‘X’ (Figure A.27).

A.11 PROPERTIES OF TRANSMITTERS [LINES BETWEEN THE POINTS]

```
14          public Camera myCamera;
15
16
17          public VectorLine mainLine;
18
19
20          public VectorLine deformedLine;
21
22
23          public List<Transform> lineTransformList;
24
25          public List<Transform> deformedLineTransformList;
26
27          public List<Vector3> forceVectorList;
28          public Texture2D frontTex;
29          public Texture2D lineTex;
30          public Texture2D backTex;
```

Figure A.28: Script for initializing key variables for drawing lines between point spheres

The Center eye anchor camera of the OVR Player Controller [46] is referred to the variable myCamera. VectorLine object is extracted from the Vectrosity [45] library and its two main variables include mainLine and deformedLine. While mainLine refers to the line drawn by the user during design modelling; deformedLine refers to the line extracted from the Matlab [51] after the design analysis (Figure A.28).

In Lines 23 and 25, list (dynamic array) of Transforms that are in identical order as mainLine and deformedLine are stored in order to keep the points updated. This list of transforms is used to store elements of points in series of connection. For instance, if a line is drawn from input to intermediate point, then the first and second elements of the list are transforms (position, rotation and scale) of input and intermediate points respectively. While a list of force vectors is being stored (Line 27), in order to transfer and receive these stored values between Unity [43] and Matlab [51]. Each element of this list comprises of force vector in X, Y, Z directions. The textures of line; front & back end cap are referred to lineTex, frontTex and backTex respectively [49] (Line 28-30). These textures belong to the data type Texture2D. The end caps are mainly used for either adding arrows or rounding the ends of a specific line (Figure A.28).

```
32          void Awake()
33          {
34              VectorLine.SetEndCap("Arrow", EndCap.Both, -1.0F, lineTex, frontTex, backTex);
35          }
36
```

Figure A.29: Script for setting the desired patterns on the edge of a line

Awake() function is used when there is a need for variable initialization before the Start function is called [48]. The line is defined within this function to maintain high speed performance by not deleting and recreating the line every frame. If there are changes to be made to the line, then the transforms of end points can be changed and thus resulting in a different line orientation. In Line 34, it can be observed that the parameters for setting an end cap include end cap type; endcap placement; textures for main line and the attached end caps. EndCap.Both is a constant within EndCap enum and signifies the inclusion of texture for both front and back ends of the line [49].

```
40          void Start () {
41              deformedLineList = new List<VectorLine>();
42              pointLists = new List<List<Vector3>>();
43              VectorLine.SetCamera3D(myCamera);
44              //Wireframe of cube
45              VectorLine line = new VectorLine("Wireframe", new List<Vector3>(), 1.0f, LineType.Discrete);
46
47              Mesh cubeMesh = ((MeshFilter)gameObject.GetComponent("MeshFilter")).mesh;
48              line.MakeWireframe(cubeMesh);
49              line.drawTransform = gameObject.transform;
50              line.Draw3DAuto();
51
52              mainLine = new VectorLine("MainLine", new List<Vector3>(), 10.0f);
53              mainLine.Draw3DAuto();
54
55              deformedLine = new VectorLine("deformedLine", new List<Vector3>(), 10.0f);
56              deformedLine.Draw3DAuto();
57
58              lineTransformList = new List<Transform>();
59              //deformedLineTransformList = new List<Transform>();
60
```

Figure A.30: Script for creating a wireframe for design domain cube and initializing key lines

The lists declared on the Lines 41 and 42 come under Start() function, since they are meant to be displayed on the Inspector of Domain gameobject as soon the game starts. This is an efficient way of declaring something that pops up during runtime. In order to draw a line effectively, Vectrosity [45] needs a reference of the camera (CenterEyeAnchor Camera) being used [49] (Line 43). In Line 45, the syntax for creating a line using Vectrosity [45], is clearly mentioned. ‘VectorLine’ is the datatype and ‘line’ is the variable. While ‘new VectorLine’ is the function for creating a new line; ‘Wireframe’ is the name of line and ‘new List<Vector3>’ creates a list of

points within the line; 1.0f refers to the thickness of the line in floating point; ‘LineType.Discrete’ refers to the Discrete type of line [49].

The variable ‘cubeMesh’ is of class type ‘Mesh’ and it is being extracted from the ‘mesh’ sub-component of the ‘MeshFilter’ Component of Domain gameobject. In this state, the cubeMesh variable is of Component class type. Hence, it is being “casted” as (MeshFilter) [Line 47]. In Lines 48-50, there are three key functions namely ‘MakeWireFrame’; ‘drawTransform’; ‘draw3DAuto’. MakeWireFrame function is used to create a desired mesh (in this case, Cube Mesh) on the line assigned to it. Function ‘drawTransform’ is used to assign the meshed line to the transform of a specific gameobject. Finally, the line is assigned to Draw3DAuto() function and it is used mainly in Update() function. In Update() function, as the camera moves around the Cube domain, the Cube gets distorted every frame, if the normal Draw() function is used. Hence, Draw3DAuto() function is always preferred if a line is continuously changing every frame (Figure A.30).

In Lines 52 and 55, the ‘mainLine’ and ‘deformedLine’ lines are created with the parameters mentioned within parenthesis. In both the lines, 10.0f refers to the thickness of the line. In Line 58, ‘lineTransformList’ variable stores a list of Transforms of various gameobjects (Figure A.30).

```
63            //Force Line
64            forceVectorList = new List<Vector3>();
65
66        }
67
68        void LateUpdate()
69        {
70            int i = 0;
71            foreach (Transform transform in lineTransformList)
72            {
73                if (transform.CompareTag("Input") || transform.CompareTag("Output") || transform.CompareTag("Intermediate") || transform.CompareTag("Fixed"))
74                {
75                    mainLine.points3[i] = transform.position;
76                    i++;
77                }
78            }
79
80            for (i = 0; i < deformedLineList.Count; i++)
81            {
82                List<Vector3> worldCoords = new List<Vector3>();
83                foreach (Vector3 point in pointLists[i])
84                {
85                    worldCoords.Add(transform.TransformPoint(point));
86                }
87                deformedLineList[i].points3 = worldCoords;
88            }
89        }
90
```

Figure A.31: Script for assigning point spheres to the key points of the line

In Line 64, ‘forceVectorList’ variable stores a list of vectors of various force elements. LateUpdate() function is very useful in situations where the transform of a line is continuously changing every frame [48]. If the Update() function is used then multiple lines will be drawn every frame, since it comprises of the lines drawn during the previous and the current frame at a single frame [49]. Hence LateUpdate() function ensures that the existing line gets converted into a new line, only after the transform is applied on it. In Lines 70-78, the foreach() loop is used and its only difference from that of a for() loop is that it iterates through a list of items. Hence, this foreach() loop iterates through all the elements of type ‘Transform’in ‘lineTransformList’ list. It can be observed that ‘transform’ is a dummy variable just like ‘i’ variable in a for() loop [Line 71]. In Line 73, ‘CompareTag’ function is used to compare the string to the ‘Tag’ transform assigned to a specific gameobject. If it turns out to be true, then the if() loop will be executed. All the points of the mainLine will have the same position coordinates (X, Y, Z) as that of the point spheres (Input, Output, Intermediate) (Figure A.31).

The for() loop is meant to be applied on each element of ‘deformedLineList’ list [Line 80]. The ‘worldCoords’ variable comprises a list of points [Line 82]. The foreach loop picks each point of pointLists() list and applies the following condition: Convert each point from its local coordinates to world coordinates using TransformPoint() function; finally the converted points are added to ‘worldCoords’ list [Lines 83-86]. Ultimately, the points in ‘worldCoords’ are assigned to ‘deformedLineList’ [Line 87] (Figure A.31).

A.12 POINT TYPE SWITCHER

The key function used to transition from one type of a point to another is the SwitchTo() function. It gets activated On Click event i.e. when the user places their thumb over the left joystick. The software behind this event handling User Interface mechanism is Virtual Reality toolkit [44]. The menu template was already available and it is customizable according to the needs of the user.

```
19      public class PointTypeSwitcher : MonoBehaviour {
20          int activeSphere;
21          enum Type:int
22          {
23              INPUT,
24              OUTPUT,
25              INTERMEDIATE,
26              FIXED,
27              DELETE,
28              FORCE,
29              FREEDOM
30          };
31
```

Figure A.32: Script for defining key constants under ENUM datatype

The enum is the keyword used for declaring collection of related constants (Line 21). A point class includes different constants such as Input, Output, Intermediate, Fixed, Delete, Force, Freedom (Figure A.32).

```
32          public GameObject fixAxesRadialMenu;
33          public GameObject forceRadialMenu;
34          public GameObject forceCanvas;
35          private void Start()
36          {
37              activeSphere = 0;
38              SwitchTo(0);
39          }
```

Figure A.33: Script for initializing key gameobjects for Radial Menus

The key GameObjects used for the graphical UI menus include FixAxesRadialMenu (FARM) and forceRadialMenu (FRM). FARM is used exclusively for assigning Fixed Axes (X, Y, Z), while drawing force vectors by free hand on a specific point (Line 32). While FRM is also used for the assigning Fixed Axes but the fundamental difference is that they provide different choices to the user (Line 33). FARM lets the user lock the force vector in a specific direction. While FRM lets the user lock the force vector and then it tracks the hand movement of the user

and locks the axis closest to the user's right controller [46]. The other gameobject is the forceCanvas (Line 34), where it represents the textbox containing force magnitudes in X, Y, Z directions (Figure A.34). In Lines 35-39, the Start() function ensures that the default condition of SwitchTo() function is assigned so that none of the points are assigned to the Touch controllers [46] automatically at the beginning of a game (Figure A.33).

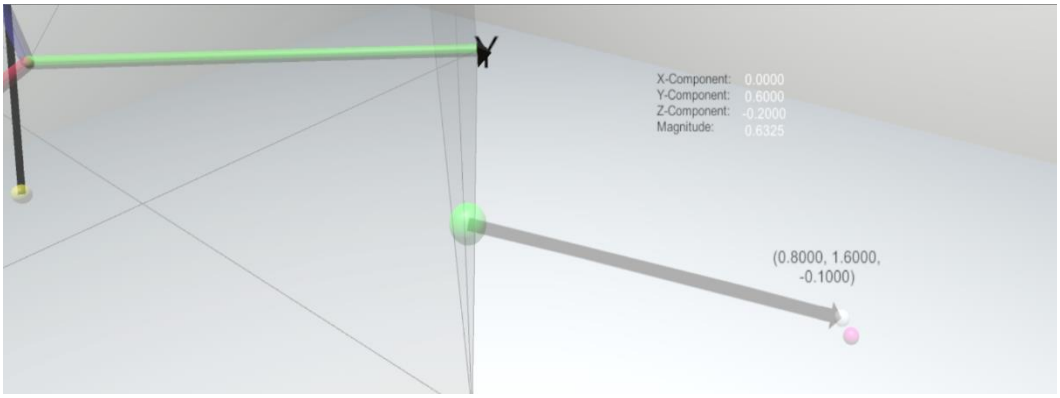


Figure A.34: Illustration of X, Y, Z components of Force magnitudes [top right corner]

```
40          public void SwitchTo(int buttonNo)
41          {
42              if(buttonNo == (int)Type.DELETE || buttonNo == (int)Type.INTERMEDIATE || buttonNo == (int)Type.FORCE)
43              {
44                  fixAxesRadialMenu.SetActive(false);
45              }
46              else
47              {
48                  fixAxesRadialMenu.SetActive(true);
49              }
50
51              if(buttonNo == (int)Type.FORCE)
52              {
53                  forceRadialMenu.SetActive(true);
54              }
55              else
56              {
57                  forceRadialMenu.SetActive(false);
58
59              }
60              if(buttonNo != (int)Type.FORCE)
61              {
62                  forceCanvas.SetActive(false);
63              }
```

Figure A.35: Script for setting activation/deactivation of various radial menus

The custom method SwitchTo() comprises of key design rules assigned to various options of different radial menus. It is similar to Switch-Case conditional statements used in other basic programming languages. The if-else statement (Lines 42-49) ensures that FARM is deactivated when the Delete, Intermediate or Force buttons are pressed by the user. One of the reasons being the Delete and Intermediate points do not need any Axis direction to be fixed; while the other reason being the Force Vector is to be assigned to FRM gameobject. In general cases, the user is aware of the magnitude and direction of the Force vector and they can follow the runtime canvas text of the Force vector and place it accurately in the domain space. As mentioned in the previous sections, SetActive() method is used to activate a specific gameobject on the execution of a specific conditional statement. FRM Gameobject, used in Lines 51-59, mainly gets activated only if the Force button is pressed. Similarly, even the forceCanvas is activated when the force button is pressed. Otherwise, it is deactivated so that it does not annoy the user with continuous Force vector updates in realtime, even when the Force option is not used (Figure A.35).

```
65            foreach (Transform joint in transform)
66            {
67                if (i == buttonNo)
68                {
69                    joint.gameObject.SetActive(true);
70                    if(!(i == (int)Type.DELETE || i == (int)Type.INTERMEDIATE || i == (int)Type.FORCE || i == (int)Type.FREEDOM))
71                    {
72                        FixedDirections fd = joint.gameObject.GetComponent(typeof(FixedDirections)) as FixedDirections;
73                        if (i == (int)Type.FIXED)
74                        {
75                            fd.fixedX = true;
76                            fd.fixedY = true;
77                            fd.fixedZ = true;
78                        }
79                        fd.reapplyColor();
80                    }
81                }
82                else
83                    joint.gameObject.SetActive(false);
84                i++;
85            }
86            activeSphere = buttonNo;
87        }
```

Figure A.36: Script for activating a specific point sphere immediately after point type is chosen

The foreach() loop looks for the transform of children of the attached gameobject and these children can be also be called joint [Line 65]. Here the gameobject refers to PointTypeSwitcher gameobject and the children include all the points right from Input to Output. As long as a specific point is chosen on the UI menu above the left controller [46] [Line 67], the selected point will be

shown immediately above the right controller [46] [Line 69]. As mentioned earlier, the fixed directions are not applied to various point spheres namely Intermediate, Delete, Force and Freedom. For the rest of the points, reapplyColor() function from Fixed Direction script is applied [Line 72-79]. Specifically, for Fixed point, all the axes are fixed using the if statement in Lines 73-78. If the value of i does not refer to any of the buttons on menu UI of Left Controller [46], then it deactivates all the point spheres [Lines 82-84]. The index of the finally activated point sphere is assigned to activeSphere variable [Line 86] (Figure A.36).

```
89          public void toggleActiveSphereFixedDir(int dir)
90          {
91              FixedDirections fd = transform.GetChild(activeSphere).gameObject.GetComponent(typeof(FixedDirections)) as FixedDirections;
92              fd.toggleDirection(dir);
93
94          }
95
96          public Vector3 GetPosition()
97          {
98              return transform.GetChild(activeSphere).transform.position;
99          }
100
```

Figure A.37: Script for getting fixed direction for a specific point sphere

The toggleActiveSphereFixedDir() function is used for calling toggleDirection() function from Fixed Direction script [Lines 89-94]. The GetPosition() function returns the X, Y, Z coordinates of the currently selected point sphere [Lines 96-99]. In Line 98, an interesting aspect to note is that the index (buttonNo/activeSphere variable) assigned in enum datatype is the same as that of the children in the PointTypeSwitcher gameobject. Hence, it is easy to extract the position coordinates of respective child of the gameobject (Figure A.37).

APPENDIX B: COPYRIGHT PERMISSION [UNITY]

Attached is the email conversation between the thesis author and attorney from Unity Technologies. This email confirms the permission granted to use Unity 3D software screenshots in this thesis.

Rks6

Apr 21, 2018 01:45 UTC

Hi,
<Very Urgent, Please reply in 1-2 days>
I am doing Masters thesis on development of virtual reality software. This VR software was developed on Unity3D game engine Version 5.6.2f1 and scripting was done in Microsoft Visual Studio. I took screenshots of the Unity Editor to explain how I created the software on Unity 3D. Could you please let me know if I need to get a written permission from Unity to use the screenshots of Unity Editor? I am asking this because I am not aware of Copyright terms and conditions of Unity 3D.

Thanks,

Ramkumar

danielm@unity3d.com

Apr 21, 2018

Hello Komanduri,

We cannot speak for Microsoft with respect to Visual Studio (you will have to contact them yourself), but screenshots of the unity editor in the thesis for only the limited educational and strictly non-commercial purpose of illustrating description of the development would be permitted, as long also as there is prominent notice given that images are "With the permission of Unity Technologies. All rights reserved." and a disclaimer in the thesis noting the following: "Names, trademarks, logos, branding, and content of Unity Technologies used with permission and are the sole property of Unity Technologies. All rights reserved. Neither this thesis nor its author is affiliated with, or endorsed or sponsored by, Unity Technologies or its affiliates." Please note that this permission would not extend to any images, trademarks, or other content displayed in the editor that are owned by a third party.

I hope this helps,

Dan McDonald
Unity Legal

APPENDIX C: COPYRIGHT PERMISSION [ASME]

The attached email confirms that there is no need for a permission from ASME for any coincidence/similarity between this thesis and the journal article DETC2018-86375 (Though the author for both thesis and journal article are same)

Email sent on 04/27/2018 at 7:10 AM:

DETC2018-86375

Philip DiVietro <DiVietroP@asme.org>
to rks6

Dear Mr. Komanduri Ranganath,

To confirm our conversation, you do not need permission from ASME to include your paper or parts of your paper in your thesis. This letter serves as your permission letter should you need one.

Best regards and all the best to you.

Sincerely,

Philip DiVietro

Philip V. DiVietro
Managing Director, Publishing
ASME
2 Park Avenue, 6th Floor
New York, NY 10016-5990
Tel 1.212.591.7696
Mobile 1.631.553.1088
divietrop@asme.org

REFERENCES

1. Kota S, Joo J, Li Z, Rodgers S. M, Sniegowksi, J. Design of Compliant Mechanisms: Applications to MEMS. Analog Integrated Circuits and Signal Processing. 2001; 29: 7-16

2. Saxena, A., and Ananthasuresh, G. K., 2000. "On an optimal property of compliant topologies". Structural and Multidisciplinary Optimization, Vol. 19(1), pp. 36–49.

3. Kota, S., Lu, K.-J., Kreiner, Z., Trease, B., Arenas, J., and Geiger, J., 2005. "Design and Application of Compliant Mechanisms for Surgical Tools," ASME Journal of Biomechanical Engineering, Vol. 127(6), pp. 981–989.

4. Krishnan, G., Kim, C., and Kota, S., 2013. "A Metric to Evaluate and Synthesize Distributed Compliant Mechanisms," Journal of Mechanical Design, Transaction of the ASME, Vol. 135(1), p. 011004.

5. Ramirez, I.A., 2014. "Pseudo-Rigid-Body Models for Approximating Spatial Compliant Mechanisms of Rectangular Cross Section" *Graduate Theses and Dissertations,* University of South Florida. http://scholarcommons.usf.edu/etd/5562

6. Howell, L.L., 2001. Compliant mechanisms. Wiley, New York.

7. Howell, L.L., Midha, A., 1995. "Parametric deflection approximations for initially curved, large-deflection beams in compliant mechanisms". Proceeding ASME Design Engineering Technical Conference, 96-DETC/MECH-1215

8. Ananthasuresh, G.K., 1994. "A new design paradigm for micro-electro-mechanical systems and investigations on the compliant mechanisms synthesis". University of Michigan, Ann Arbor, MI, ProQuest Dissertations Publishing, p. 9513290

9. Allred, T.M., 2003. Compliant mechanism suspensions.

10. Frecker, M.I., Powell, K.M., Haluck, R., 2005. "Design of a Multifunctional Compliant Instrument for Minimally Invasive Surgery". Journal of Biomechanical Engineering, Vol. 127(6), pp. 990-993.

11. Zoppi, M., Sieklicki, W., Molfino, R., 2008. “Design of a Microrobotic Wrist for Needle Laparoscopic Surgery”. Journal of Mechanical Design, Transaction of ASME, Vol. 130(10), pp. 1023061-1023068

12. Khatait, J.P., Mukherjee, S., Seth, B., 2006. “Compliant design for flapping mechanism: A minimum torque approach”. Mechanism and Machine Theory, Vol. 41(1), pp.3-16.

13. Chin, Y.-W., Lau, G.-K., 2012. ““Clicking” compliant mechanism for flapping-wing micro aerial vehicle”. Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference, 2012, p. 6385809, pp. 126-131

14. Sonmez, U., 2007. “Compliant MEMS Crash Sensor Designs: The Preliminary Simulation Results”. IEEE Intelligent Vehicles Symposium, p. 4290131, pp. 303-308.

15. Krishnan, G., and Ananthasuresh, G. K., 2008. “Evaluation and Design of Displacement Amplifying Compliant Mechanisms for Sensor Applications,” Journal of Mechanical Design, Vol. 130(10), pp. 1023041-1023049.

16. Lu, K.-J., and Kota, S., 2005, “An Effective Method of Synthesizing Compliant Adaptive Structures Using Load Path Representation,” Journal of Intelligent Material Systems and Structures, Vol. 16(4), pp. 307–317.

17. Pauly, J., Midha, A., 2004. “Improved Pseudo-Rigid-Body Model Parameter Values for End Force-Loaded Compliant Beams”. ASME Design Engineering Technical Conferences and Computers and Information in Engineering Conference Proceedings, p. 57580, pp. 1513-1517.

18. Dado, M.H., 2001. “Variable parametric pseudo-rigid-body model for large-deflection beams with end loads”. International Journal of Non-Linear Mechanics, Vol. 36(7), pp. 1123-1133.

19. Feng, Z., Yu, Y., Wang, W., 2010. “Modeling of large-deflection links for compliant mechanisms”. Frontiers of Mechanical Engineering in China, Vol. 5(3), pp. 294-301.

20. Vitellaro, G., L'Episcopo, G., Trigona, C., Ando, B., Baglio, S., 2014. “A Compliant MEMS Device for Out-of-Plane Displacements with Thermo-Electric Actuation”. Journal of Microelectromechanical Systems, Vol. 23(3), p. 6627923, pp. 661-671.

21. Sigmund O., 1997. “On the design of compliant mechanisms using topology optimization”. Mechanics of structures and machines, Vol. 25, pp. 493-524.

22. Kota, S., Hetrick, J., Li, Z., and Saggere, L., 1999, “Tailoring Unconventional Actuators Using Compliant Transmissions: Design Methods and Applications”. IEEE/ASME Transaction on Mechatronics, Vol. 4(4), pp. 396–408.

23. Aguirre, M.E., Frecker, M., 2008. “Design Innovation Size and Shape Optimization of a 1.0 mm Multifunctional Forceps-Scissors Surgical Instrument”. Journal of Medical Devices, Transaction of the ASME, Vol. 2(1), p. 015001

24. Cronin VI, J.A., Frecker, M.I., Mathew, A., 2008. “Design of a Compliant Endoscopic Suturing Instrument”. Journal of Medical Devices, Transaction of the ASME, Vol. 2(2), p. 025002

25. Hetrick, J.A. and Kota, S., 1999. “An energy formulation for parametric size and shape optimization of compliant mechanisms.” Journal of Mechanical Design, Vol. 121(2), pp. 229–234.

26. Kota, S., Hetrick, J., Osborn, R., Paul, D., Pendleton, E., Flick, P., Tilmann, C., 2003. “Design and application of compliant mechanisms for morphing aircraft structures”. Proceedings of SPIE - The International Society for Optical Engineering, Vol. 5054, pp. 24-33.

27. Frecker, M. I., Ananthasuresh, G. K., Nishiwaki, S., Kikuchi, N., and Kota, S., 1997. “Topological Synthesis of Compliant Mechanisms Using Multi-Criteria Optimization”. Journal of Mechanical Design, Vol. 119(2), pp. 238–245.

28. Joo, J., Kota, S., and Kikuchi, N., 2000. “Topological Synthesis of Compliant Mechanisms Using Linear Beam Elements”. Mechanics of Structures and Machines, Vol. 28(4), pp. 245–280.

29. Lu, K.-J., and Kota, S., 2003. “Synthesis of Shape Morphing Compliant Mechanisms Using a Load Path Representation Method”. Smart Structures and Materials, SPIE-5049, pp. 337–348.

30. Lu, K.-J., and Kota, S., 2006. “Topology and Dimensional Synthesis of Compliant Mechanisms Using Discrete Optimization”. Journal of Mechanical Design, Vol. 128(5), pp. 1080-1091.

31. Yin, L., and Ananthasuresh, G. K., 2003. “Design of distributed compliant mechanisms”. Mechanics Based Design of Structures and Machines, Vol. 31(2), pp. 151–179.

32. Krishnan, G., Kim, C., and Kota, S., 2011. “An Intrinsic Geometric Framework for the Building Block Synthesis of Single Point Compliant Mechanisms”. Journal of Mechanisms and Robotics, Vol. 3(1), p. 11001.

33. Krishnan, G., Kim, C., and Kota, S., 2013. “A Kinetostatic Formulation for Load-Flow Visualization in Compliant Mechanisms”. Journal of Mechanisms and Robotics, 5(2), p. 021007.

34. Patiballa, S.K., and Krishnan, G., 2018. “Qualitative Analysis and Conceptual Design of Planar Metamaterials With Negative Poisson’s Ratio”. Journal of Mechanisms and Robotics. Vol. 10(2), p. 021006.

35. Patiballa, S., and Krishnan, G., 2017. “Qualitative Analysis and Design of Mechanical Metamaterials”. 41st Mechanisms and Robotics Conference. Vol. 5(A), p. V05AT08A002.

36. Krishnan, G., Kim, C., and Kota, S., 2010. “Load-Transmitter Constraint Sets: Part I – An Effective Tool for Visualizing Load Flow in Compliant Mechanisms and Structures”. In Proceedings of 2010 ASME Design Engineering Technical Conference, Parts A and B, Vol. 2, pp. 563–575.

37. Krishnan, G., Kim, C., and Kota, S., 2010. “Load-Transmitter Constraint Sets: Part II- A Building Block based Methodology for the Synthesis of Compliant Mechanisms”. In Proceedings of 2010 ASME Design Engineering Technical Conference, Parts A and B, Vol. 2, pp. 577–587.

38. Kim, C.J., Kota, S., and Moon, Y.-M., 2006. “An Instant Center Approach toward the Conceptual Design of Compliant Mechanisms”. Journal of Mechanical Design, Transactions of the ASME, Vol. 128(3), pp. 542-550.

39. Blanding, D. K., 1999. Exact Constraint: Machine Design Using Kinematic Principles. ASME Press, New York.

40. Awtar, S., and Slocum, A. H., 2007. “Constraint-based design of parallel kinematic XY flexure mechanisms”. Journal of Mechanical Design, Vol. 129(8), pp. 816–830.

41. Hopkins, J. B., and Culpepper, M. L., 2010. “Synthesis of multi-degree of freedom, parallel flexure system concepts via Freedom and Constraint Topology (FACT) Part I: Principles”. Precision Engineering, Vol. 34(2), pp. 271–278.

42. Hopkins, J. B., 2010. “Design of flexure-based motion stages for mechatronic systems via freedom, actuation and constraint topologies (FACT)”. PhD thesis, Massachusetts Institute of Technology.

43. Unity, 2017. “Unity 5.6.2 Release Notes”. Unity Technologies. Retrieved from < https://unity3d.com/unity/whats-new/unity-5.6.2>

44. VRTK, 2017. “Virtual Reality Toolkit Documentation”. Sysdia Solutions Limited. Retrieved from <https://vrtoolkit.readme.io/>

45. Vectrosity, 2017. “Vectrosity Version 5.5 features”. Starscene Software. Retrieved from <https://starscenesoftware.com/vectrosity.html>

46. Oculus, 2018. “Oculus Rift features and benefits”. Oculus VR, LLC. Retrieved from <https://www.oculus.com/rift/?utm_campaign=[campaign]&utm_source=google&utm_medium=cpc&gclid=EAIaIQobChMIkOzRyuDU2gIVDp7ACh2AsgjUEAAYASAAEgLh8_D_BwE&gclsrc=aw.ds#oui-csl-rift-games=mages-tale>

47. Wikipedia, 2017. “Oculus Rift”. Wikimedia Foundation Inc. Retrieved from <https://en.wikipedia.org/wiki/Oculus_Rift>

48. Unity, 2017. “Unity User Manual 5.6”. Unity Technologies. Retrieved from <https://docs.unity3d.com/560/Documentation/Manual/>

49. Vectrosity, 2017. “Getting started with Vectrosity Version 5.5”. Starscene Software. Retrieved from https://starscenesoftware.com/files_vectrosity/Vectrosity5%20Documentation.pdf

50. Patiballa, S. K., and Krishnan, G., 2017. “Estimating Optimized Stress Bounds in Early Stage Design of Compliant Mechanisms”. Journal of Mechanical Design, Vol. 139(6), p. 062302.

51. Matlab, 2018. “Matlab features and benefits”. Mathworks, Inc. Retrieved from < https://www.mathworks.com/products/matlab.html>

52. Hololens, 2018. “Hololens features and benefits”. Microsoft, Corp. Retrieved from < https://www.microsoft.com/en-us/hololens>

53. Magic Leap One, 2018. “Magic Leap One features and benefits”. Magic Leap, Inc. Retrieved from <https://www.magicleap.com>